

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

MySQL官方专家 特邀
杜修文 宋利兵 撰稿

周彦伟
王竹峰
强昌金
著

MySQL 运维内参

MySQL、Galera、Inception
核心原理与最佳实践

◎ ACE Director携手数据库源码专家打造领域权威

◎ 集三大主流开源项目之源码剖析与实战历练于一身

◎ 凝聚社区力量，MySQL官方专家特供独家技术内幕



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

/ 作者介绍 /



—— 周彦伟 去哪儿网数据库总监 ——

Oracle MySQL ACE Director, ACMUG主席
在去哪儿网负责数据库平台的管理和维护工作。工作范围包括MySQL、Redis、HBase平台的架构设计、性能调优、日常运维及自动化运维平台设计。长期奋战于互联网行业，历经酷讯网、人人网（校内网）和去哪儿网。曾经担任人人网MySQL技术主管，负责数千规模的MySQL数据库实例的运维管理。中国MySQL用户组（ACMUG）创始人兼主席，领导和组织中国MySQL社区活动。



—— 王竹峰 去哪儿网数据库专家 ——

擅长数据库开发、数据库管理及维护，一直致力于MySQL数据库源码的研究与探索，对数据库原理及实现具有深刻的理解。曾就职于达梦数据库，多年从事数据库内核开发的工作，后转战人人网，任职高级数据库工程师，目前在去哪儿网负责MySQL源码研究与运维、数据库管理和自动化运维平台设计开发及实践工作，是Inception开源项目的作者。



—— 强昌金 去哪儿网高级DBA ——

先后就职于陌陌、去哪儿网。目前担任去哪儿网DBA，主要负责去哪儿网数据库管理平台的开发、MySQL和Redis的运维。在数据库方面，具有丰富的数据库运维和性能优化经验。

MySQL 运维内参

MySQL、Galera、Inception
核心原理与最佳实践



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书是一本介绍 MySQL 数据库知识的专业书籍,从核心原理到最佳实践,深入浅出、抽丝剥茧地进行讲解,不仅从源码和运维两个角度介绍了 MySQL 大部分重要概念和运维要点,还讲述了 MySQL 极为优秀的集群组件 Galera 的实现原理和运维经验,同时,也介绍了作者独立开发的 MySQL 审核系统 Inception 的设计、实现与功能。

本书也得到了 MySQL 官方研发团队的大力支持,两位资深专家分别介绍了 MySQL 最新的支持 NoSQL 的组件 MySQL Document Store,以及集群化组件 MySQL Group Replication 的实现原理和运维要点。

本书不仅可以作为技术管理者和架构师在设计 MySQL 相关应用和系统时的参考,还适合 MySQL 应用开发者更深入地了解和使用 MySQL。最后,作为 MySQL DBA 的必备参考,希望本书能在实际工作中对读者有所帮助。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

MySQL 运维内参: MySQL、Galera、Inception 核心原理与最佳实践 / 周彦伟, 王竹峰, 强昌金著. —北京: 电子工业出版社, 2017.6

ISBN 978-7-121-31235-9

I. ①M…II. ①周…②王…③强…III. ①SQL 语言 IV. ①TP311.138

中国版本图书馆 CIP 数据核字 (2017) 第 066498 号

策划编辑: 张春雨

责任编辑: 徐津平

印 刷: 北京京科印刷有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 40.5 字数: 884.52 千字

版 次: 2017 年 6 月第 1 版

印 次: 2017 年 6 月第 1 次印刷

定 价: 119.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 51260888-819 faq@phei.com.cn。

本书赞誉

中国君子，“穷则独善其身，达则兼善天下”。中国互联网技术从业者，也应当有这般胸怀，研习新旧技术，总结成败经验，继承开源思想，传播创新文化。很庆幸，彦伟的团队就是这样一个典型，从点滴做起，与社区共生，先做好自己，再泽被四邻。从来没有一个行业的技术趋势，如互联网这样，发展狂飙突进，门派星罗棋布，更迭日新月异。从业者要从其中海选出适合自己的方案，已是大费周章，更别说精通此道引领风尚。所以，要想跟上时代，不仅需要慧眼，更需要妙手。这本书的问世，归功于彦伟、竹峰和昌金这三位数据库老司机，不但车技娴熟，慧眼妙手，能帮他人排忧解难，而且更兼济世仁心，愿天下从业者都有医者之能。最后，衷心祝愿本书能给各位读者的职业生涯，送上一个漂亮的神助攻。

刘启荣 京东金融数据库总监

本书是理论和实践的完美结合，是去哪儿网 DB 团队知识和技术累积的结晶，有关键的理论知识，还有丰富的实践案例，同时还从源码角度来进行说明确认，使 MySQL 的爱好者知其然，更能知其所以然，在 MySQL 运维世界里是一本不可多得的好书。我受益匪浅，也希望广大读者有更大的收获。更实在的是，在去哪儿网 DB 团队负责人周彦伟先生的积极推动下，InceptionSQL 审核系统在去哪儿网的生产环境上得到了洗礼和考验，而且本书出版前该系统已开源多时，MySQL 爱好者亦可参考本书内容，然后根据自身实际的业务情况，更好地去建立或改善自有的 MySQL 自动化运维平台，方便数据库上线，减少出现错误的概率，提升运维与开发人员工作效率，解放出 DBA，使其做更有价值的事情。此外，近几年来，开源社区组织 ACMUG（中国 MySQL 用户组）在周彦伟先生的积极推动下，在主席团成员、各嘉宾的积极配合及 MySQL 爱好者的积极参与下，取得的成绩斐然，发展有目共睹，不忘初心、始终不渝地把全国范围内 MySQL 爱好者的知识和力量集合起来，共同创建一个开放、友好、免费的分享平台，让 MySQL 爱好者们在互相交流中共同进步、收获快乐。努力诠释利他才是生命的真正意义，欢迎加入开源世界并贡献力量！敬为书序。

田发明 央视网系统运维部高级经理

阅读完毕彦伟兄发来的全书摘选章节：InnoDB 索引实现原理、揭秘独特的两次写、Galera Cluster 的设计与实现和 Inception 诞生记，通过阅读这四个章节的部分内容，可以窥猜全书的技术文采，阅读后让人感受到作者是一位有丰富故事的 DBA，同时又是有着产品思路和源码经验之士。值得数据架构师、DBA 等仔细研读。

金官丁 热璞科技创始人兼 CTO

很高兴看见 MySQL 领域又能有一本新书发行，同样作为一名作者的我，很明白其中的艰辛与付出。多年前，我研究 MySQL 时遇到的最大问题就是市面上可参考的书籍太少。相信随着大数据与互联网+时代的来临，MySQL 数据库将渗透到各个领域。本书作者是我熟悉的业界资深运维与开发专家，相信本书能从全方位的角度让大家认识 MySQL 数据库。

姜承尧 网易数据库负责人

认识彦伟是三年前，在去哪儿网一间还未装修、布满网线的会议室里。第一次见面，彦伟便给我留下了敢于尝试、乐于分享的印象。过去这些年，彦伟一直在和各种不同的数据库打交道，见证了 MySQL 从一个小型的关系型数据库发展成为各大互联网企业的核心数据库的过程。他本人也一直保持着对新技术的执着。授人以鱼不如授人以渔，本着交流和分享的精神，本书作者将多年实践中积累的点滴经验整理分享出来，具有绝对的实践和指导意义。数据库的发展离不开运维的责任感，以匠心耕耘专业，这是一本有责任感、有专业精神、诚意满满之作。

阳学仕 宝存科技董事长兼首席执行官、创始人

推荐序 1

周彦伟，我们又称呼他为盐味。在浙江大学同窗七年的时间里，盐味给人的印象是聪明又很认真，幽默而又内向，专注却不失好奇。所以，在他毕业之后去了一家软件公司的时候，我们都觉得他会成为一名合格的程序员。但他却总有意外之举——先是在写了五年程序之后，改行做了 DBA；然后又以中国 MySQL 用户组主席的身份，把 MySQL 社区做得风风火火、红遍全国；紧接着又被 Oracle 授予象征 Oracle 和 MySQL DBA 至高荣誉的 ACE Director；同时，他又领导了一个关于 MySQL 审核的开源项目 Inception。作为他的大学同学和一名计算机技术教育从业者，我真心为他高兴。

最近得知，盐味的新书《MySQL 运维内参》即将完稿出版，我深知写书的艰辛与不易，这是需要巨大的恒心和毅力，付出比常人更多的心血和汗水才能完成的，非常敬佩他和他的创作团队。

我曾在同济大学计算机科学与技术系参与过数据库课程的教学工作，遍观《MySQL 运维内参》的内容，其对数据库系统原理和基础实现的讲解非常细致和认真，结合基本功能和源码实现的讲解又能让读者更清晰地理解 MySQL 技术，而理论与实践的结合更能维护论点、说明原理。相信此书能够帮助数据库特别是 MySQL 数据库学习者和使用者在学习工作中百尺竿头、更进一步。

王瀚漓

同济大学计算机科学与技术系教授、博导

推荐序 2

MySQL 是一个非常优秀的开源关系数据库管理系统。我第一次接触 MySQL 是在 2002 年，令业界称道的开源和高性能两大特性吸引我去尝试了解它。虽然后来由于种种原因没有在 MySQL 领域深耕，但一直很关注它的发展。MySQL 的快速普及，除了得益于互联网的高速发展，其支持事务和行级并发的 InnoDB 存储引擎也功不可没，InnoDB 设计上的简洁精美及其高性能令人叹服，值得数据库应用开发者、数据库管理员和数据库爱好者花时间去研读。学习它的设计思想，能帮助我们了解关系型数据库最核心的运作原理，我后来的工作深受其启发，获益良多！

《MySQL 运维内参》不仅仅是一本关于 MySQL 运维的书。在第一部分 MySQL 篇，作者介绍了 MySQL 服务器线程的启动和运行原理，并用相当大的篇幅详细介绍了 InnoDB 的数据字典对象和数据存储的细节，特别是 InnoDB 无处不在的索引和 B+ 树算法，以及保证物理文件存储结构合法性的物理事务概念，这一概念对于关系型数据库系统的 ACID 特性至关重要。此外，还介绍了基于 Binlog 的复制，可用于经典的 MySQL 高可用场景。MySQL 特有的嵌入式存储引擎架构在逻辑复制中可能引发 ROWID 问题，另外 InnoDB 半物理半逻辑的 REDO 方式，可能导致潜在的数据页断裂，因此需要引入特有的二次写入。这些 MySQL/InnoDB 深层的特性和机制，随着 MySQL 篇的展开，都一一呈现在读者面前。阅读这些章节，如同和一个在数据库实现方面有深厚功底的专家对话，能让我们了解 MySQL/InnoDB 的优势和局限，从而在实际工作中扬长避短，发挥 MySQL 及支撑它的硬件潜能。

保证数据安全有效是数据库运维的基本职责，作者以亲身经历，讲述了 MySQL 数据库在运行中可能碰到的故障和排查方式，以及如何规避可能的风险等。这些经验对于我们在实际工作中处理超长事务、死锁现象等都有很好的借鉴意义。

本书的第二篇讲述了高可用 Galera 集群的原理、实现细节，以及使用中可能碰到的问题和解决方式。Galera 是一种基于复制的允许多点读/写的高可用集群系统，而其他 MySQL 的高可用架构基本都是主从式的，这使得 Galera 在处理业务的负载均衡方面具有明显的优势。考虑到作者一手打造了国内最大的 Galera 集群，并已经投入到高并发大流量的互联网生产环境，本书关于 Galera 原理和运维的相关描述的权威性是毋庸置疑的，因此本书也是难得的 Galera 学习和使用的参考资料。

MySQL 的小巧灵活及过于快速的成长，也让它付出了代价。MySQL 擅长快速执行充分调优的相对简单的 SQL，而处理复杂 SQL 的能力很弱，因此甄别那些有可能会拖跨 MySQL 的劣质 SQL 语句（这个问题对于其他数据库系统也存在，只是程度不同）成为每一个 MySQL DBA 的重要任务，而 Inception 正是帮助 DBA 自动处理这一工作的利器，能极大提升这方面的效率。本书的作者同时也是 Inception 软件的作者，他们无私地把 Inception 这一成果贡献给了社区，并在本书中讲述了 Inception 的起源、功能特性和使用方法。Inception 的实现利用了很多编译知识，如果能结合 Inception 的源码阅读本书的相关章节，一定能收获更多。

我虽然不是 MySQL 领域的专家，但是数据库相关的很多原理知识都是相通的，有幸能在出版之前阅读到本书，我深切感受到其语言表达和技术内容之美。现在推荐给大家，希望本书能对大家的工作、学习有所帮助。

韩朱忠

达梦数据库高级副总经理

推荐序 3

在 DB-Engines 网站上，维护着一个数据库流行度积分榜，最近两年的积分榜前两位一直是 Oracle 和 MySQL，并且两者的积分已经相当接近，我查看了一下目前的分数，Oracle 是 1416，而 MySQL 是 1366，由此可见 MySQL 的流行（2017 年 1 月 17 日数据）。

一个开源的关系型数据库能够挑战 Oracle 数据库十几年来雄霸天下的位置，这本身就是开源领域的巨大成功。而一个产品的成功，不仅仅是靠技术层面的安全、稳定和高效，更重要的是要有活跃的社区和生态圈。在中国 MySQL 生态的构建过程中，彦伟的贡献有目共睹，他在完成自我的技术成长之后，又率先拉起大旗创立了 ACMUG 用户组，并且踏踏实实地在全国范围进行活动组织和技术分享，极大地活跃和带动了 MySQL 社区的发展，彦伟也因此而成为了国内第三位 MySQL 方向的 Oracle ACE 总监。

所谓同声相应、同气相求。在彦伟的影响下，一批 MySQL 俊彦汇聚到去哪儿网的数据库团队，其中就包括本书另两位作者——年轻专家竹峰和昌金。在技术上的实践和积累需要长期的底蕴和实干，彦伟的团队近年在去哪儿网的践行也尤其值得关注。现在，他们将实践多年的积累汇聚成书，与行业里的同仁分享，这对大家实在是难得而宝贵的财富。

我阅读了本书的部分章节，对比了 MySQL 的两次写与 Oracle 数据库相关实现上的异同，感觉颇为受益；而 Inception 作为 MySQL 的 SQL 审核产品，与云和恩墨的 z3 有异曲同工之妙，从 Inception 的诞生到开源，我一直都在关注，好的思路和产品，也必然是殊途同归的。我一直认为，在开发测试阶段强化 SQL 质量审核，是 DevOps 在数据库领域的最佳落地点，也是 DBA 们将踩过的坑和开发经验进行分享的最具价值的呈现，防患于未然的事前审核优化，才是对企业和业务负责任的态度。现在，作者们把 Inception 的来龙去脉呈现出来，一定会让很多 DBA 们感同身受，并开始学习和借鉴。

一本好书，十年磨砺！愿读者朋友们能够体味其中甘苦，一同尽饮 DBA 们带来的佳酿！

而至于无穷回味、激发创新、转折演绎，则要靠各位去谱写新的篇章！

盖国强

云和恩墨创始人，Oracle ACE 总监，ACOUG 主席

推荐序 4

由于 ACMUG (中国 MySQL 用户组) 的缘故, 我很早就认识了彦伟, 并且意气相投、一见如故。他是 ACMUG 的创始人兼主席, 同时也是一名运维经验丰富的 DBA, 他曾经分享过在人人网鼎盛时期以区区几名 DBA 应对 MySQL 巨大访问量的经历, 也谈起过在去哪儿网主导的针对电子商务和交易的 MySQL 运维架构的革新和 SQL 审核工具 Inception 的创作。从 UGC 类型的人人网到电商类型的去哪儿网, 这对 DBA 本身就是一个进阶。从使用开源工具到自己创作并开源工具, 这更是一个巨大的进步。

王竹峰就是 Inception 的主要开发者, 他的数据库理论和源码功底非常深厚, 而他边运维边进行源码开发和研究的工作方式更加促进了他对 MySQL 整个体系架构原理的理解, 带着问题看源码, 装着源码解决问题, 通过源码解读 MySQL 的各种特性, 这已经成了竹峰的标签。

我是以运维 DBA 的角色开始我的职业生涯的, 经历了长期一线实战的积累和锻炼, 随后才慢慢转为数据库内核开发。即便是现在, 我跟阿里云的其他同事也会经常处理一线运维问题, 通过对这些问题的处理, 我们积累了经验和需求, 再反馈到数据库源码中去, 这是我们不断前进的原动力。一线运维经验与数据库理论和源码相结合的重要性, 我有深刻的体会。

得知彦伟愿意牵头把他和团队这几年的经验和技能通过书的形式写出来分享, 我非常欣喜并期待, 这是对广大 MySQL DBA 最好的礼物。

彭立勋

MariaDB 基金会成员, ORACLE MySQL ACE

推荐序 5

自从 2015 年 10 月 MySQL 5.7 GA, 以及 2016 年 9 月 8.0 DMR 版本启动后, 我们可以看到 Oracle 官方明显加大了对 MySQL 的开发力度, 努力把 MySQL 打造成一个全明星产品, 让我们对 MySQL 未来的发展更加坚定了信心。

作为一个从业多年的 MySQL DBA, 我从未停止探索理解 MySQL 中各方面的技术实现细节, 包括最重要的 Server 层及 InnoDB 存储引擎。此外, 我也关注 MySQL 的各种高可用实现方案, 以及作为 DBA 非常迫切需要的自动化管理平台。

2016 年 12 月, MySQL 5.7 正式发布了 MySQL Group Replication, 我们终于可以用上这个官方的多节点同时写入高性能架构的方案了。当然, 新事物总是需要有一个阶段才能完善, 还不能大规模上线使用。那么, 现阶段如果有这样的需求怎么办呢? 毫无疑问, 肯定是选择 Galera Cluster 方案。

如果有这样一个工具, 能帮助 DBA 完成 SQL 上线前的审核, 审核通过后能自动上线, 万一反悔或误操作还能“倒带”(回滚), 相信您一定会很期待吧! 如果这个工具还开源了, 是不是更兴奋了呢? 没错, 这就是 Inception! 它完全可以帮您实现这些梦想, 让 DBA 无须再痛苦地做 SQL 审计和上线执行。

在过去的几年里, 周彦伟先生、吴炳锡先生和我一起打造了国内最有影响力的 MySQL 用户组织 ACMUG。从和周彦伟的协作中能感受到他对技术细节的拿捏把控。因此我们也有理由相信, 由周彦伟先生、王竹峰先生等人主导编写的这本 MySQL 大作将会引爆今年的 MySQL 技术圈。同时我们也要感谢本书所有作者的辛苦编撰和无私的经验分享。

叶金荣

知数堂培训联合创始人, ORACLE MySQL ACE

推荐序 6

我和彦伟是老朋友了。最近得知彦伟在写一本叫作《MySQL 运维内参》的书籍，感到若合一契。

互联网刚开始进入中国的时候，国内数据库市场一直被昂贵的商业数据库统治，在一些大型企业、机构尤为明显。但最近几年，以 MySQL 为代表的一批开源数据库已经在慢慢成长，足以应付复杂的使用场景，也足以在那些要求高可用、高一致性、高安全性的领域施展拳脚。国内在 MySQL 方面的人才也快速涌现。这个时候，我们都感到需要一些有行业经验的人站出来，把成熟的 MySQL 使用经验分享给大家，让 MySQL 能够真正独当一面，发挥更大的价值，成为众多企业的选择。

腾讯云在金融、保险、工业、政府、游戏互联网应用等多个领域和从业人员有过一些交流，他们也感觉缺乏运维方面的书籍来培训和指导他们的员工。彦伟这次的专著应运而生，既是行业的需要，也是他作为 ACMUG 创始人的责任。

彦伟书中内容翔实，不仅介绍了使用方法，也对 InnoDB 的原理、主从复制，以及业界内的一些不同做法做了介绍。文中介绍的 Inception，与腾讯云的备份审计系统颇有相似之处。我们也经常在一起探讨云数据库的架构与实现，希望本书能给读者带来新的启发，为 MySQL 带来新的动力。

祝百万

腾讯云数据库技术负责人

自序 1

十年前，我有幸加入了陈华、吴世春创办的酷讯网，开启了我的互联网职业生涯，同时也让我开始接触到开源世界的件件瑰宝，尤其是 MySQL。终于，两年后，我正式成为了一名 MySQL DBA。这要感谢当时如日中天、被誉为中国 Facebook 的校内网和我的良师益友刘启荣先生，让我有机会和空间，在 MySQL 的世界里自由翱翔并得其所哉。

从事 MySQL DBA 工作是我一直以来引以为豪的事情。我一向认为 DBA 是一个与众不同的职业，如果要把人分成两类的话，那么有一种分法就是，一类是 DBA，另一类是非 DBA。

DBA 是什么？

- 有点像 SA。但在业务层面上比 SA 事多，主要是牵扯的人多，嬉笑怒骂，皆成文章。
- 有点像 DEV。但总是会有冲突，多数情况下发生在双方对数据库使用的看法和优化上。我们只好自己开发一个 Inception 给 DEV 用。
- 有点像 PM。设计库，优化表，处处都是艺术。
- 有点像 CTO。你以为我说的是首席建表官？DBA 不过是操着鸡毛蒜皮的心啊。
- 有点像出租车司机。一向不待见那种不等改完表甩手就走的人，等待是一种美德，善始善终方显英雄本色。
- 有点像消防队员。网站挂了，可能跟 DB 没关系，但 DBA 一定要出现；DB 挂了，一定跟网站有关系，DBA 也一定要出现。
- 有点像银行点钞员。数据就是钱啊，只是都不是自己的，但职业道德最为重要。
- 有点像养孩子。当爹又当妈，哪个实例伺候不好，都会出乱子。

DBA 是服务型职业，服务好了，没你什么事；服务不好，事情就大了。

DBA 承载着一个重要的角色，有着特殊的职责和使命，一个优秀的 DBA 其实是非常难得的。DBA 应该具备怎样的能力呢？我曾经提出过“DBA 精神”：责任心、服务心、沟通心、学习心、进取心和分享心。

DBA 精神是责任心的体现

维护数据库数据的安全和完整是管理员的首要责任。作为一个 DBA，在管理数据库的过程中，要把数据库看作自己的财产、儿女和身体的一部分。此种职责，需要你像呵护自己的眼睛一样去照顾你所维护的数据库。时刻去想，有没有做应该有的备份，有没有加应该有的监控，有没有做必须要做的安全权限限制。一旦出了问题，有没有第一时间去分析和解决问题。这就像自己的眼睛，一旦出现红肿病态，怎会有人视而不见？

DBA 精神是服务心的体现

现代互联网对 DB 的需求，需要支撑业务持续稳定的运行和源源不断的变更，这要求 DBA 有 7×24 小时的服务精神。这是一个服务性职业，需要随时随地响应来自各方面的各种需求。其一，人的需求，业务要发展，DB 就会有变更，只有 DBA 与开发人员紧密配合，才能顺畅高效地完成工作。其二，事的需求，DB 是一个动态的系统，不断地运转就会不断地面临新问题，机器故障、磁盘报警、内存不足、CPU 过载等。这是一个脆弱的系统，它不会对 DBA 的迟钝有半点仁慈，一旦有问题，任性的宕机是必然的结果。其三，心的需求，DBA 应该发自内心地主动去对 DB 不断地做优化，这种优化，可以在结构上，可以在架构上，可以在业务逻辑上，关键在于有没有用心。说得通俗一点，“没事找事”。

DBA 精神是沟通心的体现

DBA 的工作，不是封闭的科学研究和孤傲的英雄主义，不管是主动地优化，还是被动地接受任务，都需要与人做不断的沟通与交流。如果不懂业务流程，不知道数据的来龙去脉、轻重缓急，那就谈不上数据库表的设计和优化。而数据库的优化，最大的进展往往来自对业务逻辑的优化。这需要运用良好的沟通心态和技巧，深入了解业务流程和架构。与人沟通，是 DBA 获取知识和信息的主要途径，需要做到能与人沟通和会与人沟通。在生活工作中，人的个性是千差万别的，面对形形色色的沟通对象，充分调动对方的积极性，进行愉快有效的交流，也是 DBA 需要掌握的一门技术。

DBA 精神是学习心的体现

DB 在技术体系中是一个承上启下的中间环节，它运行在物理硬件和操作系统之上，同时承载着上层各种各样的业务逻辑的调用。对 DB 的运维和管理，不仅仅需要掌握数据库自身的基础知识，还需要知己知彼，去了解它上下游的知识和特性，才能保证这个系统的稳定和优化。所以，一个优秀的 DBA 需要了解你所使用的硬件，这包括磁盘的特性、网络的布局、内存的使用和 CPU 的处理能力等；需要了解 DB 所运行的操作系统的知识，它是怎样调度 I/O 的，怎样管理内存的，怎样优化文件的；需要了解上层业务是怎样调用数据库的，SQL 是怎样写的，看懂了业务逻辑的程序才能明白某些 SQL 是不是多余的，某些 SQL 是不是可以优化。MySQL 官方和 MariaDB 官方，都已经针对硬件开始在官方源码的基础上做自动优化了，同时 MySQL 官方还提供了自己集成于业务端的高可用方案，这都是开源数据库进步

发展的必然结果。同时，由于开源数据库的盛行，针对数据库自身代码的学习和研究，也逐渐成为优秀 DBA 的必修课了。

DBA 精神是进取心的体现

随时发现问题、分析问题、解决问题。通过问题现象，依赖自己的经验和知识，同时探索未知的知识去解决现实中的问题，在这个过程中，也是自己积累经验和不断成长的过程。相对而言，解决问题本身并不重要，重要的是在此过程中探索解决问题的方法并总结所获取的经验，DBA 的职业优势也在于此，同时，主动花费心思与精力去不断追求问题的优解和技术的极致，也是 DBA 应尽的职责。

DBA 精神是分享心的体现

这些年，我们一直在倡导做一件事情，那就是提倡 DBA 在开源社区持续做技术分享。我们做了 ACMUG，全称是 China MySQL User Group。A 可以认为是 All、About，或者是 Active，它代表了所有关心 MySQL 及其相关技术的积极参与分享活动的人。在过去的数年里，ACMUG 已经组织了上百场技术分享，分享者都是工作在一线的 DBA 或数据库技术爱好者，通过参与这个活动，大家都切切实实地得到收益，个人在技术上也取得了巨大的进步。通过分享，大家学到了技术、开拓了视野、展开了思路，当然也交到了朋友。我们希望能通过这样的活动，将 DBA 精神延续下去，让更多人受益，让更多技术普及。

本书的写作，正是出于分享的目的。我们依靠开源软件和技术实现了自己的职业理想和人生价值。饮水思源，投桃报李，我们愿意把自己在工作中的点滴积累分享出来，以期能帮助到更多的人。

写书，其实并不是那么轻松。周围的好多朋友都有这样的想法，甚至已经付出了巨大的努力与心血，但最终并未能如愿，非常地遗憾。一方面，互联网是一个以快为王道的行业，大家平时的工作并不轻松，尤其是 DBA，7×24 小时的服务要求你要时刻准备着投入工作。另一方面，开源社区也是日新月异、一日千里，稍一迁延，所述的题材和内容也就落伍了。

下面一段引文是我在 2015 年 7 月为将要开始写作的这本书写的一篇自序，这是当时要写的内容的一个轮廓，现在看来已经大相径庭了。

去哪儿网是一个做线上旅游产品的网站，主营线上机票和酒店，以及其他旅游相关的产品，例如火车票、度假线路和旅游攻略，目的地生活，接送机场、景点、火车站的专车，景点门票等。发展到现在，去哪儿网已经算是一个纯粹的电子商务网站了，而在网上的各种业务逻辑和操作，大部分都是用 MySQL 作为后台数据库支撑的，这也包括交易的支付业务和账务系统。要完成这些，除了依靠强大的前端业务逻辑及丰富的流水日志外，对 MySQL 运维管理也是极大的挑战，这也使得去哪儿网的 MySQL DBA 得到了极大的历练。

在工作过程中，为了支撑业务的发展，围绕着 MySQL 运维这个主题思想做了很多事情，大致来说，可以分为以下几个方面。

其一，定义《去哪儿网 MySQL 数据库开发规范》。这主要是针对开发人员的数据库操作规范，定义这个规范的主要目的是为了更方便我们规范地使用 MySQL 数据库，在可行的情况下，充分利用数据库的优点去解决我们的业务需求，同时最大可能地规避已知 MySQL 的不足之处。另外还有一些约定俗成的规范，例如表名、字段名要用小写字母，这完全是使用习惯的规范，大家完全不必太在意，可以根据自己的习惯调整，但我建议，如果在同一家公司内部，还是统一起来比较好。

其二，开发《InceptionSQL 审核系统》。为了最大程度地保障线上数据库安全，我们规定，所有的线上 SQL 必须通过 DBA 的审核之后，才能上线执行，因为在高并发、高负载的系统中，任何一条不优化的 SQL 都有可能引起数据库负载过高，从而导致数据库实例挂掉。去哪儿网是一个快速发展的公司，各种业务需求变化多端，平时 DBA 审核 SQL 的压力非常大，再加上我们定义了自己的开发规范，这在人工审核的过程中很可能出现一些问题，例如审核不全面，尺度不统一，判断不正确等。为了改善这一状况，提高 DBA 的工作效率，加快项目上线的速度，我们花大力气开发了一套 InceptionSQL 审核系统，这套系统能完成 DBA 平时审核过程中的大部分人工工作，极大地提升了工作效率，也大大提高了审核的正确性，目前已经是 DBA 工作过程中不可或缺的有力帮手了。

其三，优化数据库架构，提升数据库的可用性。去哪儿网是实时的在线交易网站，它对交易数据的一致性和可用性要求非常高，在这个目标上，我们付出了极大的努力。长期以来，去哪儿网的 MySQL 架构是 MMM，MMM 是一个比较实用的 MySQL 集群方案，在早期的 MySQL 运维中，一度被很多人采用，在 MySQL 领域是一个使用比较广泛的架构。但是随着技术的发展和业务对数据库服务能力需求的提升，MMM 已经渐渐不能满足我们的需要了，特别地，它有一些致命的缺陷，可能会导致业务完全无法使用。我们花费了巨大的精力去探索什么才是适合自身的 MySQL 架构方案。最终，我们选定了基于 Galera 同步的 Percona XtraDB Cluster (PXC) 方案。在此方案中，我们选定了 PXC 为底层数据库存储架构，同时加上自主开发的分布式哨兵监控系统及连接池管理模块，形成了一套完整的数据库高可用架构方案，这套方案在线上运行良好，极大地满足了线上业务的高可用和高一致性的需求。

另外，由于基于 PXC 的数据库方案是高一致性的，它对可用性做了必要的牺牲，但在业务中，有些地方是不太计较一致性的，或者可以接受相当程度的延时。为了满足此类业务的需求，我们在保留 PXC 的同时，另起了一个新的方案 QMHA，它起源于 MHA，但我们不满足 MHA 的监控方式及线上切换的缺陷，同时，觉得自己的分布式哨兵监控系统工作得实在太好了，所以决定在 MHA 的思路，用组件来完善这

个架构，它的底层是数据库的 Master-Slave Replication，用分布式哨兵作为监控工具，同时利用 MySQL 5.6 之后的 GTID 和 Semi-Sync-Replication 这些特性，来保证数据的最终一致性。

第四，自动化运维。台上一分钟，台下十年功。如果安全的线上运维是台上表演的话，那么围绕着 MySQL 的自动化运维和数据安全所做的工作就是台下功了。我们在这方面做了不少工作，其中值得一提的是 MySQL 备份系统，完善的备份是整个数据库系统必须要有的重要一环。它在某种意义上也是数据库高可用的一部分，一旦有线上误操作或其他意外，备份数据会救公司一命。我们使用 Percona XtraBackup 自己定制了一套完善的备份系统，可以作为一些场景需求的参考。

另外，为了在工作中能快速完成集群的部署，我们定制了自己的集群部署平台。它可以实现分钟级的 MySQL 集群部署，这里需要指出的是，我们的集群结构很复杂，它除了传统的启动 MySQL 实例外，还需要管理监控哨兵、Zookeeper 资源、集群 namespace 资源，以及一些常规的备份和监控功能。

最后，为了方便管理我们的数据库集群，完成复杂的线上节点切换和维护，我们实现了数据库集群管理平台，让 DBA 在线上操作时，可以摆脱复杂的运维步骤，大大简化了操作流程，同时也有效防止了误操作的可能，提高了运维操作的安全性。

随着去哪儿网 DB 团队的壮大，我们的知识和技术也在慢慢积累。这些知识平时散布在各种 Wiki、Doc、Blog 及内部邮件中，系统性和可读性都非常差，非常不利于知识的传播和传承，特别是最近新人入职，对他们的培训也要花费很大的精力。我不禁想，如果能把这些知识整理出来，做成一本书，岂不两全其美。如果去哪儿网之外的技术同仁们也对此感兴趣，我也乐于把它分享出来，供大家在工作中参考。如果能对大家有所帮助，也算是我们对技术社区的点滴回报，真是善莫大焉。

MySQL 是历史悠久的开源数据库，有着非常强大的社区和人数巨多的拥趸，这里面卧虎藏龙、大牛辈出。由于我们知识范围有限，如果本书中有疏漏或不妥之处，还望见谅，我们会积极改进，不断进步。

周彦伟

2015 年 7 月 21 日于北京

亲爱的读者会发现，有很多文中提到的内容在本书中并未涉及。真实的情况是，近两年来，我们一边工作，一边偷空写作，写写停停，停停写写，随着时间的流逝，很多内容已经不再适合放入本书，例如我们对于 MMM 和 MHA 的使用方案和运维经验；另外，在实践中我们也发现，有些技术和方案是紧密联系业务的，事关公司敏感信息的内容也无法放入本书。于是，我们又有了写写删删、删删写写的痛苦经历。技术书籍的时效性对于我们有切肤之痛，曾经有多少次，本书几乎走到了夭折的边缘。

这里，要衷心感谢一下我最亲密的战友王竹峰先生，与之相识、相知真乃人生一大快事。竹峰的存在，不仅帮我实现了自己诸多的技术设想，也帮我坚定了去做很多事情的信心。在本书的写作过程中，竹峰的坚持与毅力鼓舞了大家，竹峰的贡献是本书能够出版的关键。本书的另一位作者强昌金，在工作和写作中所表现出来的初生之犊的冲劲和孜孜不倦的精神，也极大地推动了本书的完成。

给读者带来最实用的内容是我们一直追求的目标。我们能力有限、学识浅薄，只能花出十二分的气力，写一些自己熟悉和擅长的内容呈现给读者。对于没有把握的内容，宁可缺失，也不能草草充数了事。这些年，我们最熟悉的东西有三种，那就是本书中介绍的 MySQL、Galera 和 Inception。MySQL 是本书的根本，我们使用了多年，而且竹峰在 MySQL 源码上的钻研心得也极大地提高了本书的含金量。我们在去哪儿网所推行的基于 Galera 的 PXC 方案，算是近年来最先进的 MySQL 集群化方案之一了，它的数据一致性和高可用性特别适合去哪儿网这种涉及交易和电子商务的服务型业务，经过三年的实践，它帮我们支撑了大部分业务需求，使我们受益匪浅。而 Inception，则是我们送给所有 MySQL DBA 和 MySQL 相关开发人员的礼物，Inception 的出现，把 DBA 审核 SQL、操作 SQL 的效率提升了一个量级，也为数据库操作的安全性和规范性提供了行业参考，作为 Inception 的设计、开发、使用及推广者，我们不能不去提它。

而对于已经不再适应时代发展和技术潮流的内容，我们也绝不手软。MMM 和 MHA 的部分曾经写得很辛苦，而临到最后，我们还是决心把它们删去，如果有希望了解这方面知识的读者，请自行查找吧。但还是要奉劝大家，MySQL 集群化的时代已经到来，基于 Galera 的 Percona XtraDB Cluster 和 MariaDB Cluster 及最新的 MySQL Group Replication 才是上上之选。

MySQL 已经进入了 MySQL 5.7 时代，并且 MySQL 8.0 也已经呱呱落地。我们也高度重视在这方面的匹配。虽然开始写书的时候还在用 MySQL 5.6 版本，但是到了后期，除了某些原理性的源码解析，对于版本差别没那么敏感之外，我们还努力去兼容新版本的特性。尤为荣幸的是，来自 MySQL 官方资深的技术顾问杜修文先生和 MySQL 一线开发团队的宋利兵先生愿意以特邀撰稿人的身份特别为我们的书写了一部分内容，这是 MySQL 最新版本相关功能的权威解读。

我们在书中努力加入了很多在平时工作过程中碰到的案例及经验总结，也算是本书的一个特色吧，希望这些能在实战中切切实实地帮助到读者。

最后，还是想把上面几年前说过的话再重复一遍：MySQL 是历史悠久的开源数据库，有着非常强大的社区和人数巨多的拥趸，这里面卧虎藏龙、大牛辈出。由于我们知识范围有限，如果本书中有疏漏或不妥之处，还望见谅，我们会积极改进，不断进步。

周彦伟

2017 年 1 月 25 日于北京

自序 2

北漂帝都，转眼间已四年有余。这四年间，有两个人对我有着至关重要的影响，我必须先行致谢。

首先感谢我的好友周彦伟先生。入京以来一直是他带领着我学习、工作，让我有巨大的进步。一直想写本书，但苦于知识水平有限，客观条件也不允许，一直没能如愿。在他的帮助下，才得此机遇，圆了这个梦，一个我们共同的梦，再次感谢。

其次要感谢我的爱人田丽芳女士。不管是生活中，还是工作中，她的秀外慧中、深明大义及勤奋严谨的作风，都让我感动。在本书的写作过程中，她也做了很大的贡献，帮助做了内容方面的校正，这也得益于她对 MySQL 的热爱，并能意志坚定地选择加入 MySQL DBA 这个大的行列中来，真心感谢她。

互联网发展至今，开源软件已经深入人心，并且受到广泛的支持和响应，很多公司在使用开源软件的同时也输出了一些好的开源产品。MySQL 作为当今世界上最受欢迎的开源数据库产品之一，在很多互联网企业里起到了不可或缺的作用。由于 MySQL 的诸多特性，比如开源免费、灵活、轻量简单且越来越多的企业开始使用 MySQL，在业界诞生了一大批相关从业者，他们研究 MySQL 的原理，探讨 MySQL 的架构，完善 MySQL 的运维，丰富 MySQL 的工具，促进 MySQL 的发展，我们称这些人为 MySQL DBA，而本人也是其中之一，深感荣幸。

怎样才能成为一名合格的 MySQL DBA？在写作过程中，我一直在思考这个问题。受能力和见识所限，也许本书不能很好地回答这个问题，但根据我个人的成长经历，还是有几点感悟可以跟大家分享。

1. MySQL DBA，是一类操作型工种，必须要经常动手，想要快速成长，需要经历很多有计划的或真实的线上问题处理才可以。一个优秀的 DBA，需要见多识广，达到处事不惊、遇事不乱的状态，这样才能在问题出现时，果断出击，快速恢复服务。经验是 DBA 最大的财富。
2. MySQL DBA，也是一类服务型工种，属于服务行业。我们的使命就是要最大程度地保证业务所依赖的后端数据库的稳定性，并且提供最长的可服务时间。基于此，DBA 都要尽力去培养一颗“为人民服务”的心，时刻做好在第一线战斗的准备。

3. 本人最初从事的是数据库内核开发工作，相对 MySQL DBA 来说，这属于两个层面，内核开发是以 MySQL 内部实现原理及方法为视角的，而 MySQL DBA 是以如何使用好它为视角的。如果有机会，可以将这二者做一个完美结合，或者角色互换，将会有完全不同的效果，因为只有实际使用了，才可以产生实际的需求，才能促进 MySQL 更好地发展。而做过开发之后再做运维工作，毫无疑问会非常轻松，并且有希望发展到目无全牛的境界。

作为一名有缘既做数据库开发又做数据库运维的 DBA，在撰写本书的时候就是本着让每一位对 MySQL 感兴趣的读者可以由表及里、深入浅出地去理解 MySQL 的初衷来构想的。在本书中，我会简单介绍 MySQL 的运行原理、InnoDB 存储引擎包括哪些重点模块，以及各个模块之间又是如何很好地配合来完成用户提交请求的等内容。但在讲述这些实现原理时，我往往会更加强调方法及深层次的理解，而不局限于其本身，同时还会强调读者自身是如何思考的，在学习过程中，必须要时刻伴随自己的思考想象，加上 MySQL 的自身实现，才能不断地对其加深理解，做到融会贯通。

本书在讲述 MySQL 时，不会是面面俱到的，所以书名也不是类似“MySQL 宝典”之类的。本书主要讲述从 MySQL DBA 的角度来看，哪些对运维来说是重要的，哪些对业务来说是高效的，并且他们都该如何理解，比如日志、索引等章节。

本书的内容，还包括了在去哪儿网经历了多年实践的 Galera Cluster。Galera 无疑是 MySQL 界的福音，因为它改变了长久以来，MySQL 异步复制导致经常修复数据的局面，并且因为其多点写入的特性，给 MySQL 的应用和架构设计带来了非常大的发挥空间。本书要做的，就是基于我们长达三年多的使用经验，并且立足于对其深入的理解，来讲述 Galera Cluster 是如何实现的，如何支持了多点写入，冲突是如何解决的，有哪些问题且如何避免，诸如此类都会有所涉及。

很多人只听到了 Galera Cluster 可能出现的问题，再加上目前使用者及相关文档书籍比较少，导致出现了恶性循环，而本书其中的一个目的就是将这些谜团一一揭开，还原一个真正强大好用的 Galera Cluster。万事不可能完美，我想表达的观点是，当问题不再是问题时，还有什么能阻止你对它的青睐？

本书还包括了一部分特殊的内容，即我们的开源产品 Inception。根据目前的反馈，它在业界已经被广泛使用。这也是在我从开发转行做了 DBA 之后，将开发与运维结合的产物，也是我们使用开源产品之余，向业界贡献的一款开源产品。很多 DBA 反映，由于它的出现，MySQL DBA 的工作瞬时有了翻天覆地的变化，并且轻松了很多，是广为传颂的业界良心产品。作为 Inception 的开发者，我深感荣幸，同时也诚惶诚恐、战战兢兢，只能尽自己最大的努力，争取把它做得更好、更强大，以感谢广大支持者的厚爱。Inception 已经开源，其手册也已经在网络上广泛传播，而本书所要讲述的，不局限于手册内容，更包含了对 Inception 的一种理解、它对我们的意义及一种设计理念。

本书还包括了很多案例，这是在长期运维工作中，我和同事不断积累的一些我们认为值得学习的方法及知识点。

在本书中，很多章节都涉及了有关源码的解读和分析，由于不同章节，写作时间不完全一样，其中一些章节，尽量保持了与最新版本源码同步，比如讲述关于“MySQL 源码编译方法”的章节就是，因为是编译方法的学习，采用的是最新的源码。

而有些涉及代码的部分，主要是用来讲述 MySQL 或者 InnoDB 的内部实现原理。在大多数情况下，即使代码稍有改变，原理也不会有太多变化，所以很多还是使用了 MySQL 5.6 版本的源码，我的写作初衷是想要大家尽可能地去理解，做到融会贯通，并且主要讲述的是学习方法，而不局限于其本身，更不是照本宣科地只是学习某个版本的源码而已。

所以针对出现了源码内容的部分章节，还是要根据自己的兴趣，带着自己的问题，根据相关章节中提供的思路，追根溯源，不断地往复。对每一个问题，首先要有自己的初步想象，在这个问题解决之后，将自己的想象与其真实的实现方法对比，不断地校正自己的想象，从而产生新的问题，循环迭代，不断提升自己对 MySQL 的理解。

我热爱学习，也热爱分享。在生活中，不管是好的想法，还是好的资源，都喜欢去分享。能以个人微不足道之力，给周边朋友或是业界，甚至整个社会带来一些福利，贡献一点力量，足矣！

最后，在学习、生活方面，我想与大家分享一段话作为结语：

盖士人读书，第一要有志，第二要有识，第三要有恒。有志则不甘为下流；有识则知学问无尽，不敢以一得自足，如河伯之观海，如井蛙之窥天，皆无识者也；有恒则断无不成之事。此三者缺一不可。

——曾国藩

王竹峰

2017 年 1 月 19 日于北京

前言

MySQL 是开源世界里面一颗璀璨的明珠，是最流行的关系型开源数据库。关于 MySQL 的著作浩如烟海、充栋盈车。如何才能写出一本有特色内容的书呢？本书独辟蹊径，从运维和实践相结合的角度，分专题和知识点逐一讲解。用源码去解释 MySQL 的知识点，分析碰到的问题，这就是本书的特色。

本书内容所涉及的知识点的选择，建立在我们多年工作经验的基础之上。在平时维护 MySQL 的过程中，觉得需要引起注意或需要弄清楚的，就把它分享出来，希望能管中窥豹，帮助到读者朋友。

本书总体分三个部分。

第一部分是 MySQL。

在这一部分中，我们介绍了 MySQL 大部分常见知识点的概念和原理，以及运维经验。所谓源码面前，了无秘密。我们从 MySQL 源码入门开始，介绍源码结构、启动过程、创建连接、表对象缓存等，希望读者在研究 MySQL 的时候，能充分借助 MySQL 源码开源的优势，从源码出发，深入理解 MySQL 的精髓。当然，限于篇幅，我们只能抛砖引玉，适可而止。期望聪明的读者能打好基础，登堂入室。

本部分会重点介绍 InnoDB 的相关概念，从源码目录结构，到 InnoDB 体系架构及其数据字典、数据存储结构、索引的实现原理、两次写（DoubleWrite）、日志管理机制、InnoDB 记录格式等，都进行了深度的剖析，这块内容比较复杂，也凝聚了我们巨大的心血，期望能帮助读者理解其中的相关概念。

随着 MySQL 5.7 的成熟及 MySQL 8.0 的发布，一大波 MySQL 新技术迎面扑来，我们也不会放过这次学习的机会。值得称道的是，本书中包含了来自两位 MySQL 官方团队的专家特地给我们供稿的内容，分别详细讲述了 MySQL 对 JSON 的支持：MySQL Group Replication 和 MySQL Document Store。另外，我们也关注了 MySQL 的 GTID、SYS Schema、Semi-Sync Replication 等，通过对它们详细的描述，相信本书能为广大读者在快速熟悉和掌握 MySQL 新版本中出现的功能的学习助上一臂之力。

在讲述各个知识点的同时，也会结合工作过程中的一些经验，跟大家分享自己的心得，例如我们详细介绍了 Percona XtraBackup，也分享了在使用它的过程中碰到的问题。类似地，还有快速删除大表的案例、死锁的案例、处理很多文件时碰到的问题等。

最后，值得一提的是，我们还结合各个知识点，对 MySQL 如何保障数据库安全做了比较详细的总结，希望这会改变人们对开源数据库是否可靠这样问题的一些有争议的看法。

第二部分是 Galera。

Galera 是去哪儿网架构 MySQL 高可用的技术基础，我们选择了以 Galera Cluster 为基础的 Percona XtraDB Cluster 为技术原型，自主研发了针对去哪儿网这种以交易为主需求的电子商务网站的 MySQL 高可用架构方案，目前已经在线使用了三年，可能也是国内最大的 Galera 集群，我们也很荣幸能成为 MySQL 集群化的早期践行者。

本部分首先介绍了 Galera Cluster 的设计与实现，让读者能从宏观层面理解它。当然，虽然说是宏观，但是根据本书写作的主导思想，也不免会从源码的角度去剖析其架构、理解其精华。Galera 提供了非常丰富的参数让使用者去灵活地设置，我们也做了重点介绍和解析。

在准备工作完成之后，开始讲述 Galera 的重要知识点：验证方法、消息传送、GCache 实现原理、SST/IST 细节、Donor/Desynced 详解、并发控制、FlowControl 及 grastate.dat 文件揭秘等。仔细研读了这些内容之后，相信读者对 Galera 的理解会更上一层楼。

实践是本书的另一个特色。所以在 Galera 部分也加入了不少的实践案例，例如业务更新变慢的根由溯源、手动构建集群死锁、从库的转移等，通过这些内容分享工作中的一些心得，希望读者能够从中受益。

第三部分是 Inception。

这是我们从零做出来的一个开源项目。Inception 已经开源，其手册也已经在网络上广泛传播，而本书所要讲述的，更是对 Inception 的一种理解，以及它的意义和设计理念。

关于它的诞生、安装与使用，以及支持的选项、结果集和命令集等，我们都会介绍。同时，值得注意的是，我们特地安排了一节 Inception 的彩蛋，里面讲述了几个 Inception 的特殊功能，它们已经成为了日常工作中不可或缺的重要工具。

如何阅读本书

基于本书的目的，不是为了帮助你了解整个数据库的所有内容，如果想要了解整个 MySQL 数据库的所有内容，在线手册无疑是最好的帮手。而本书只关注了在工作中总结出来的对工作有用的重点内容。所谓的重点内容很多，它们之间可以不相关，也可以相关，所以行文组织是比较自由的，相应的阅读模式也可以比较自由。

书中的 MySQL 部分，其讲到的案例都是非常独立的，与其他章节没有什么关系，所以可以采取跳跃式阅读。对于一些讲述原理的内容，如果不同章节中有相关联的部分，在内容中都会有类型参照关于“某某某”的内容来引导阅读，所以也不需要刻意遵守章节顺序。

书中的 Galera 部分，大部分是在讲述它的实现原理，对于这些内容，最好是按照书中的顺序阅读，这样才可以最好、最快速地理解它们。而后面会讲述一些操作方法及案例等内容，都是基于前面的原理的，案例之间没有相关性，可以采取跳跃式的阅读方法，找到自己感兴趣的内容。

书中的 Inception 部分，讲述了如何诞生、如何使用、如何设计及所支持的功能等内容，这基本也是有顺序的，建议按照书中顺序来阅读。

读者对象

如果你是一名技术管理者，期望本书能帮你建立对 MySQL 数据库的信心，助你完成数据库方向的技术体系建设，同时也能帮助你了解 MySQL 的原理和架构。其中的知识要点，如果在面试中使用的话，应该能帮你找到你所渴求的那 1% 的顶尖人才。

如果你是一名架构师，期望本书能帮你在 MySQL 架构上大显身手，本书中介绍的 MySQL Replication、Semi-Sync Replication、Group Replication、Galera Cluster，几乎囊括了所有 MySQL 架构的基础，一定有一款适合你。

如果你是一名开发者，期望本书能帮助你在开发过程中，充分理解 MySQL 的原理，设计出合理的数据库表和索引，更好地利用 MySQL 的优势，避开 MySQL 的劣势，合理使用这个优秀的开源数据库。

如果你是一名 DBA，那么真心恭喜你，本书的知识能助你成为那 1% 的顶尖人才。

勘误和支持

由于我们对 MySQL 及相关技术的认知水平有限，以及在书写过程中可能存在一些疏忽，书中可能还存在一些不尽如人意的地方，或是不够完美还需要改进的地方，抑或是存在一些错误等问题。我们希望广大读者朋友们能指出其中的问题并留下您的宝贵建议或意见，我们会不断改进，不断完善，在此先感谢大家。

如果有朋友有任何关于本书的问题，或者建议意见等，想要与我们联系，可以发邮件到 mysql@dbace.club，我们会尽己所能及时回应大家。

我们会在我们的公众号上随时发布本书勘误细节和最新技术进展，同时也会把相关重要意见不定期地结集发布，为了保持随时沟通并获取最新的改进资料，可以扫描右侧二维码关注我们的公众号：formysql。



致谢

感谢去哪儿网这个大平台。有机会去实践个人理想是一件值得庆幸的事情，而我们恰恰是幸运的。作为 DBA，去哪儿网从业务和规模两个方面给我们提供了无比宽阔的平台。去哪儿网是典型的电子商务网站，并且业务种类繁多，场景复杂，这可以让我们有大把的机会去实践各种业务需求，特别是电子商务网站对数据一致性要求的严谨和服务可用性要求的苛刻，可以说是不断在逼迫我们积极创新、精益求精，这无疑促进了我们自身的进步。同时，去哪儿网作为名列前茅的行业平台，有庞大的用户群和访问量，这对数据库的性能和并发都提出了很高的要求，在这种情况下，DBA 工作的紧迫和艰苦表现得淋漓尽致。宝剑锋从砥砺出，梅花香自苦寒来。多年以后我们发现，在这样的环境下，我们进步了。

在去哪儿网，要特别感谢吴永强先生和甘泉先生。他们先后见证了去哪儿网 DBA 团队的组建和成熟，是他们在背后的鼎力支持，才让我们有机会风风火火地折腾一场，完成了 DBA 技术体系的实践。同时，也要感谢去哪儿网 DBA 团队。如果说我们在 MySQL 上还有那么一点点小小的成果的话，那么这个成果一定是属于这个团队每一个人的，这是大家共同努力的结果。这个团队目前的成员有：李坤、孙凯、徐庶、郑平奎、高岩、强昌金、朴昌俊、许子文、沈彦、蒲聪、吕尚、李勇、黄勇、王竹峰、周彦伟。

感谢我们的老朋友杜修文先生和宋利兵先生，他们也是我们众多 MySQL 官方团队朋友的代表。在我们多年使用 MySQL 的过程中，他们曾经给予过我们无数次的指导和帮助，是我们学好 MySQL、用好 MySQL 的良师益友。在这本书的写作过程中，他们又热心相助，牺牲自己的业余时间，为我们呈现了 MySQL 在 NoSQL 和 Group Replication 方面的最新技术细节，无论对本书，还是对于广大的读者，都是意外之喜。

在我们学习和使用 MySQL 的过程中，个人知识和经验的积累及解决问题的思路大部分都来自朋友们的指点和教诲，我们要感谢帮助过我们的朋友们，特别是一起参与 ACMUG 社区活动的朋友们，限于篇幅，我们不能点名逐一致谢，就以 ACMUG 代替吧，感谢 ACMUG。

特别地，感谢为本书作推荐的作者：甘泉先生、王瀚漓先生、韩朱忠先生、盖国强先生、刘启荣先生、田发明先生、彭立勋先生、金官丁先生、叶金荣先生、祝百万先生、姜承尧先生、阳学仕先生，诸位师长和朋友的倾情推荐给了我们巨大的信心和勇气，我们表示衷心的感谢。

由于我们几人出身工科，文字功底粗陋，同时在技术上也不可避免地破绽百出。这里要特别感谢参与本书校对的李坤、田丽芳、高岩，是他们不厌其烦的耐心核对和明察秋毫的细心校正，才使得本书不至于佶屈聱牙、不堪卒读。

最后，感谢本书的读者，你们的支持是我们最大的动力，谢谢你们！

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 扫码直达本书页面。

- **下载资源**: 本书如提供示例代码及资源文件, 均可在 [下载资源](#) 处下载。
- **提交勘误**: 您对书中内容的修改意见可在 [提交勘误](#) 处提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **交流互动**: 在页面下方 [读者评论](#) 处留下您的疑问或观点, 与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/31235>



目录

第一部分 MySQL 篇

1	MySQL 源代码入门	5			
	MySQL 源代码的组织结构 ...	5			
	Linux 下的编译	7			
	安装 MySQL 库	11			
	MySQL 5.7 权限处理	12			InnoDB 存储引擎启动与关闭 . 54
2	MySQL 启动过程	14			InnoDB 存储引擎的启动 ... 55
3	连接的生命与使命	22			InnoDB 存储引擎的关闭 ... 59
	用户连接线程创建	22	6	InnoDB 数据字典	66
	MySQL 处理请求	27		背景	66
	总结	31		系统表结构	67
4	MySQL 表对象缓存	32		字典表加载	69
	表结构的实现原理	32		Rowid 管理	75
	涉及的参数变量	42		总结	76
	优缺点总结	45	7	InnoDB 数据存储空间	77
	存在的问题	45		表空间文件组成结构	78
5	InnoDB 初探	47		段	78
	InnoDB 的源代码目录结构 ...	47		簇	79
	InnoDB 存储引擎文件组织 ...	49		页面	80
	InnoDB 体系结构	52		段、簇、页面组织结构	80
			8	InnoDB 索引实现原理	91
				背景	91
				B+ 树及 B 树的区别	91
				索引的设计	92
				聚簇索引和二级索引	94

二级索引指针	95	日志的意义	157
神奇的 B+ 树网络	97	日志记录格式	159
InnoDB 索引的插入过程	99	日志刷盘时机	163
一个页面至少要存储几条记录	107	REDO 日志恢复	165
页面结构管理	110	数据库回滚	180
文件管理头信息	111	数据库 UNDO 段管理	181
页面头信息	113	数据库 UNDO 日志记录格式	186
最小记录和最大记录	115	回滚时刻	188
页面数据空间管理	116	总结	189
经典的槽管理	117		
页面尾部	121	12 MySQL 5.7 中崭新的 MySQL sys Schema	191
页面重组	122	Performance Schema 的改进	191
索引页面的回收	123	sys Schema 介绍	192
9 InnoDB 记录格式	125	sys Schema 视图摘要	193
背景	125	sys Schema 重点视图与应	
从源码入手了解行格式	126	用场景	194
总结	133	使用风险	198
10 揭秘独特的两次写	134	总结	198
单一页面刷盘	135	13 方便的 MySQL GTID	199
批量页面刷盘	136	GTID 相关概念	200
两次写组织结构	137	什么是 GTID	200
批量刷盘两次写实现原理	138	GTID 集合	200
两次写的作用	139	GTID 生命周期	201
发散思维	139	GTID 的维护	202
总结	139	gtid_executed 表	202
11 InnoDB 日志管理机制	141	gtid_executed 表压缩	203
InnoDB Buffer Pool	141	GTID 搭建主从	204
REDO LOG 日志文件管理的		搭建主从时, 需要注意的	
用途	147	MySQL 参数	204
MTR InnoDB 物理事务	150	开启 GTID	205
		搭建主从	205

使用 GTID 案例总结	206	17 MySQL 快速删除大表	252
如何跳过一个 GTID	206	背景	252
利用 GTID 模式快速改变		问题分析	252
主从复制关系	210	案例解决	255
在线将传统模式复制改为		发散思维	256
GTID 模式复制	211	总结	257
在线将 GTID 模式复制改		18 两条不同的插入语句导致的死	
为传统模式复制	212	锁	258
GTID 的限制	213	背景	258
14 MySQL 半同步复制	215	问题分析	260
半同步特性	215	发散思维	265
半同步主库端	217	总结	265
半同步从库端	222	19 MySQL 在并发删除同一行数据	
半同步实现	224	时导致死锁的分析	266
插件安装	228	背景	266
半同步自动开关	228	问题分析	267
15 MySQL 5.7 多线程复制原理	229	发散思维	276
背景	229	总结	276
行之有效的延迟优化方法	229	20 参数 SQL_SLAVE_SKIP_COUNTER	
MySQL 5.6 的多线程复制	230	的奥秘	277
MySQL 5.7 的多线程复制	231	21 Binlog 中的时间戳	280
ordered commit	232	背景	280
多线程复制分发原理	238	问题分析	280
异常故障恢复	239	发散思维	284
16 大量 MySQL 表导致服务变慢的		事务中的事件顺序	285
问题	244	问题延伸	285
背景	244	show processlist 中的 Time	288
问题分析	245	总结	291
案例解决	247	22 InnoDB 中 Rowid 对 Binlog 的影	
总结	251	响	292
		背景	292

问题分析	292	MySQL 集群安全保证	327
总结	297	传统的主从模式如何保证	
23 MySQL 备份: Percona Xtra-Backup 的原理与实践	298	数据库安全	328
备份背景及类型	298	Semi_Sync Replication 方式	
认识 Percona XtraBackup	299	的复制	332
XtraBackup 的工作流程	300	MySQL 集群化如何保证数	
XtraBackup 的备份原理	302	据库安全	333
XtraBackup 需要的权限	304	总结	335
innobackupex 常用的备份选		26 MySQL 性能拾遗	337
项说明	304	适当的数据文件大小	337
XtraBackup 备份实践	307	碎片空洞问题	338
全量备份	307	设计问题	338
增量备份	308	合理设计表结构	339
并行备份	311	冗余存储	339
其他备份	311	拆分存储	341
案例实践与心得	312	重复存储	342
建议与提醒	314	特别提醒	342
24 MySQL 分库分表	315	正确使用索引	343
分库分表的种类	315	MySQL 系统参数	345
分库分表的原则	317	内存和 CPU	347
分库分表实现	320	磁盘的革命	348
数据库层的实现	320	云中漫步	351
业务层的实现	322	总结	354
25 MySQL 数据安全	323	27 MySQL Group Replication	356
单机安全	324	Group Replication 概述	356
集群安全	324	组的概念	357
备份安全	324	多主复制	358
MySQL 实例安全保证	325	单独的通信机制	358
Double Write	325	Group Replication 服务模式	359
REDO LOG	325	单主模式	359
		多主模式	360
		服务模式的配置	363

Binlog Event 的多线程执行 . . .	363
group_replication_applier	
通道	363
基于主键的并行执行	364
搭建 Group Replication 复制	
环境	364
MySQL 的参数设置	365
Group Replication 插件的使	
用	366
Group Replication 插件的基	
本参数设置	367
Group Replication 的数据库	
用户	368
Group Replication 组初始化 .	368
新成员加入组	369
Group Replication 的高可用性 .	370
组内成员数量的变化	371
强制移除故障成员	371
Group Replication 的监控	371
Group Replication 的基本原理 .	374
状态机复制	374
分布式的状态机复制	375
分布式的高可用数据库	376
深入理解 Group Replication	
中事务的执行过程	377
本地事务控制模块	378
成员间的通信模块	379

全局事务认证模块	382
异地事务执行模块	387
事务流程的总结	387
深入理解成员加入组的过程 . .	389
组视图	389
加入组时视图的切换	390
View_change_log_event	391
恢复	391

28 MySQL Document Store 面面观 . . 394

新的 JSON 数据类型和 JSON	
函数	395
JSON 数据类型	395
JSON 函数详解	396
JSON 函数的运用	408
MySQL X Plugin 和 X Protocol .	410
支持 NoSQL 所做的努力 . . .	410
安装 MySQL X Plugin	411
MySQL Shell	411
安装 MySQL Shell	412
运行 MySQL Shell	413
在 MySQL Shell 中操作	
JSON 文档	414
用脚本执行 MySQL Shell . .	420
X DevAPI	422
总结	425
参考资料	425

第二部分 Galera 篇

- 29 Galera Cluster 的设计与实现** 430
- Galera Cluster 的优点 430
 - Galera 的引入 431
 - Galera 接口 433
 - 总结 442
- 30 Galera 参数解析** 443
- 状态参数 443
 - 变量参数 448
- 31 Galera 的验证方法** 455
- Binlog 与 Galera 的关系 455
 - 验证方法 456
- 32 Galera 的消息传送** 458
- 33 GCache 实现原理** 461
- 配置参数 461
 - 实现原理 462
 - 发散思维 465
- 34 大话 SST/IST 细节** 467
- 初始化节点环境 468
 - 连接到集群并且做 SST/IST 469
 - 如何提供增量数据 477
 - 总结 477
- 35 Donor/Desynced 详解** 479
- 实现方式 480
 - 意义何在 480
 - 问答环节 481
- 36 Galera 的并发控制机制** 482
- 数据复制 482
 - 写集验证 483
 - 写集 APPLY 483
 - 事务 Commit 484
- 37 Galera 的流量控制** 485
- 流量控制的定义 485
 - 流量控制的实现原理及影响 486
 - 两个问题 488
- 38 Galera Cluster 影响单节点执行效率的因素** 489
- 单点验证 489
 - 并发控制 490
 - 等待 GTID 490
 - 总结 490
- 39 grastate.dat 文件揭秘** 491
- 引子 491
 - 分析研究 492
 - 总结 492
- 40 Galera Cluster 从库的转移** 494
- 没有开启 Server 级 GTID 的情况 495
 - 开启了 GTID (server 级) 的情况 496
 - 总结 500
- 41 Galera Cluster 节点与其从库的随意转换** 501
- 背景 501
 - 从节点向 PXC 节点的转换 502

	PXC 节点向异步从节点的转 换	504	用 Binlog 来代替触发器	517
42	业务更新慢, 不是由 Galera 引起 的	505	表名交换	520
43	在线改表引发的 Galera Cluster 集群死锁	516	Galera Cluster 中的问题	524
	背景	516	一个有趣的实验	526
			解决方案	531
			总结	532

第三部分 Inception 篇

44	Inception 诞生记	537	47	Inception 的备份回滚	562
	关于 SQL 审核	537		备份存储架构	562
	半自动化方法	537		备份所需条件	567
	人肉法	540	48	审核规范	568
	不满现状的追求	542		支持的语句类型	568
	何谓 Inception	543		公共检查项	569
45	Inception 安装与使用	545		插入语句检查项	570
	下载和编译	545		更新、删除语句检查项	570
	启动配置	547		表属性检查项	570
	线上配置需求	548		列属性检查项	571
	需要额外注意的点	549		索引属性检查项	572
	使用方法	549		修改表语句检查项	573
	举例说明	551		总结	573
	环境变量的设置	553	49	参数变量	574
46	支持选项	555		语法和变量	574
	选项说明	555		注意事项	578
	DDL 与 DML 语句分离	559	50	友好的结果集	580
	小技巧	560		结果集结构	580
				总结	583

51 命令集语句..... 584

- 远程信息获取..... 584
- 显示本地全部变量..... 586
- 显示本地某个变量..... 586
- 设置本地变量..... 587
- 显示 OSC 执行进度..... 588
- 查看当前 processlist..... 589

52 Inception 的彩蛋..... 591

- 对 OSC 的支持..... 591
- 可选的 OSC 参数..... 591

查看 OSC 的执行进度..... 593

中止 OSC 的执行..... 596

查看所有 OSC 执行信息... 597

Inception 对 SQL 执行情况的

统计..... 597

打印语法树..... 599

53 Inception 设计..... 609

Inception 之源..... 609

Inception 执行流程..... 611



第一部分 MySQL 篇



内参〔2017〕1号

MySQL 的创始人叫 Michael “Monty” Widenius，大家都亲切地称呼他为 Monty，并尊他为“MySQL 之父”。

1975 年，Monty 拥有了一台属于自己的可编程计算器，德州仪器的 Ti-58，512 字节的内存。

1978 年，Monty 在父亲资助了一半的钱的情况下，买了属于他的第一台个人电脑 ABC 80，配置是 4MHz、8KB ROM 和 8KB RAM，另一半的钱是他在赫尔辛基的马路上帮人铺沥青赚到的。虽然当时已经有了 Apple II，只是 Monty 觉得 ABC 80 运行 Basic 语言更快，因此选择了 ABC 80，否则，如今留着一个 Apple II 这样的古董，也是价值连城了。

1980 年，Monty 的电脑更新到了 ABC 800，配置是 32KB 内存和 4MHz 的处理器。

1981 年，Monty 在简陋的 ABC 800 上用 Basic 语言写下了 MySQL 的第一行代码，那个时候还在叫 Unireg，也就是 MySQL 的前身。

1983 年，Monty 有了一台 DS90。在这台装有 UNIX 系统的电脑上，他第一次用 C 语言重写了 Unireg。这个时候，他的电脑内存只有 2MB。

1994 年，Monty 跟他的伙伴们把 SQL 交互加入到 Unireg，并重新命名为 MySQL，真正的 MySQL 诞生了，“My”是取自 Monty 大女儿的名字。

1995 年，MySQL 在双许可（商业 + 开发）的开源协议下发布，双许可协议为开源软件的商业化探索出了一条新路，也可以更好地支撑开源软件的发展。

1996 年，Monty 还在只有 24MB 内存、40MHz 的 Sun SPARC Station 上工作。Monty 是一个非常喜欢接受挑战的人，并且对自己的能力充满了自信。在硬件资源紧张的情况下，他在数据库里面选择了多线程模型来处理客户端的并发请求，这要比多进程模型节约很多资源。这也奠定了 MySQL 不同于 Oracle 及其他一些主流数据库的一个重要特性，这个特性对后来 MySQL 的发展有着深远的影响。

2007 年，Monty 离开 MySQL，并于 2009 年开始开发全新的完全支持 MySQL 的 MariaDB，“Maria”是他小女儿的名字。由于 MariaDB 是在 MySQL 的基础上重写并完全支持 MySQL 的，所以业内在提到 MySQL 的时候，一般情况下也包含了 MariaDB，同样还有 Percona Server。本书以下内容皆是如此。

从上世纪 90 年代后期开始，互联网的热潮开始席卷全球，也为 MySQL 的发展提供了广阔的空间。特别是基于交互的社交、电子商务及金融互联网的发展，对传统的数据库服务能力提出了挑战。高并发、高性能、高可用、轻资源、易维护和易扩展的需求，促进了 MySQL 的长足发展。MySQL 流行的原因可以归结为以下几点。

简单

尽管 MySQL 源码的入门门槛很高，很多人不能深入其中，但是 MySQL 的使用还是非常简单的，任何稍微有 IT 背景的技术人员都可以无师自通地参照文档安装运行和使用 MySQL，这几乎没有什么门槛。同时，MySQL 支持大部分 ANSI SQL-92，略有经验的使用者都能完成基本的操作需求。

开源

开源意味着更加安全，代码就摆在那里，无数的技术爱好者一起来审核程序，一起修补问题，这让使用者非常放心。同时，开源也带来了免费。从上世纪 90 年代末一直持续到现在，互联网产业的快速兴起及发展让各大互联网企业对免费数据库的需求非常迫切。免费，也让规模化部署的需求成为可能。要知道很多大型互联网公司的 MySQL 集群都是数以千计或者万计的，如果是按照传统商业数据库的收费模式，费用会很难让人接受。免费，让数据库大规模使用成为可能，也提升了互联网交互式服务的质量。

复制

MySQL 从 3.23.15 这个版本开始，支持了 Replication，能够支持 MySQL 使用者搭建 Master-Slave 架构，把数据准实时地从一个实例同步到另一个实例。具备这一功能的时间是在 2000 年之前。要知道，其他一些开源数据库是在最近几年里才开始支持这样的功能，这对于数据库使用者来说，不管是在线备份还是读写分离，或者负载均衡到多个读库，意义实在是太大了。在 2010 年之前，一个 Master、多个 Slave 的部署情况随处可见。我见过有 12 个 Slave 的情形，可以设想，如果没有 Replication 功能，要用数据库处理百万甚至是亿万规模的查询负载是非常困难的。

引擎

MySQL 不同于其他多数数据库之处是它对插件式存储引擎的支持，这是一个开放的设计，有点“兼容并包，海纳百川”的感觉。熟悉 MySQL 的人随便就能报出几种 MySQL 存储引擎的名字，MyISAM、InnoDB、NDB、TokuDB 等。而 MySQL 里最常用的，也是奠定了 MySQL 开源数据库之王地位的 InnoDB，并不是 Monty 或者他的伙伴们开发的。插件式存储引擎的设计，让 InnoDB 及其他存储引擎轻松接入到 MySQL Server，集百家之长，这样 MySQL 就有了无限的活力和竞争力以保持其长盛不衰。

支持

Monty 说在早期 MySQL 刚刚推出的时候，他亲自写了 30000 封邮件来帮助人们使用 MySQL。这样持之以恒、不辞劳苦、不厌其烦的精神让人赞叹。我们自己也维护了一个开源数据库项目——InceptionSQL 审核。在过去的一年多里，我们也遇到了很多很多使用者提问的问题，这种感受，真的是只有经历过才能体会其中的付出。

合作

MySQL 在发布的早期，就广泛地和其他社区合作，PHP 和 Perl 的开发者都很愿意去传播 MySQL 的技术和相关新闻，很多 Linux 版本都会预装 MySQL。LAMP (Linux、Apache、MySQL、PHP) 一度成为站长或者开发者的标配。MySQL 在各种合作中逐步深入人心，遍地开花。

社区

社区对 MySQL 的贡献功不可没。MySQL 流行的一个重要原因就是人们常说的社区力量强大。社区为 MySQL 贡献了架构方案、运维工具、技术文档、宣传普及，乃至专业人才。放眼望去，不管是专业的数据库服务团队，例如著名的 Percona，还是著名的技术型公司，例如 Google、Facebook，都在使用 MySQL 的过程中不断地给 MySQL 贡献新的功能和工具，帮助 MySQL 更加成熟和稳定。在国内，这几年我们一直在组织 ACMUG（中国 MySQL/MariaDB 用户组、China MySQL/MariaDB User Group），它汇聚了中国最顶尖的 MySQL 及其周边技术人才，同时得到了腾讯、阿里巴巴等公司同行的支持和积极参与。公司有竞争，然而技术却是无界的，大家广泛交流，互通有无，共同促进中国 MySQL 技术的传播和进步。目前，ACMUG 及 ACMUG 所组织的技术交流活动得到了 MySQL 所属的 Oracle 公司、MariaDB 所属的 MariaDB 基金会及 MySQL 和 MariaDB 之父 Monty 先生本人的认可，我们会继续前行，为中国的开源数据库做出贡献。

迄今为止，MySQL 应该是最成功的开源数据库了。在本书中，我们不打算从 0 开始介绍 MySQL 的方方面面，仅从工作经验和需求出发，抽取在此工作中总结出来的精华部分分享于此，希望能够帮助读者解决相关的问题。

MySQL 源代码入门

先从 MySQL 源代码开始 MySQL 的历程吧，这是理解 MySQL 架构体系和技术细节最靠谱的途径。以往无数的经验表明，在源码面前，任何的总结经验和惯用习惯都显得非常苍白无力，源码才是揭开真相的唯一途径，即便是官方的发布文档，也不可避免地有这样那样的疏漏或者跟现有版本不同步之处。在现实的生产实践中，有很多文档解决不了的问题，都通过查看源代码找到了答案。所以，即便是一个 MySQL 运维 DBA，了解一些源代码也是很有必要的。这一节不打算详细去说具体的细节，相信只要把组织结构和编译调试方法说明白了，聪明的读者都会顺着这个思路去慢慢展开的。同时，这也就是说，学习源码没有特别好的捷径，一点一点地去啃，坚持不懈，才能融会贯通。当然，在以后的章节中，会不断地拿出源码片段跟大家一起分析 MySQL 的某些功能原理。

MySQL 源代码的组织结构

MySQL 数据库是开源的，它的源代码可以在其官网上直接获得。在其官网页面，默认下载的是其最新版本的源代码。在编写本节时，最新版本是 MySQL 5.7.16。如果有 MySQL 账户，可以登录下载，当然也可以跳过登录直接下载。

下载之后的文件一般是 `mysql-5.7.16.tar.gz`（以 5.7.16 为例）的打包文件，解压之后，可以看到最原始的代码结构，如图 1.1 所示。

```
zhufeng@zhufengmac:/data/workspace/mysql-server$ ls
```

BUILD	cmake	libevent	plugin	support-files
CMakeLists.txt	cmd-line-utils	libmysql	rapid	testclients
COPYING	config.h.cmake	libmysqld	regex	unittest
Docs	configure.cmake	libservices	scripts	vio
Doxyfile-perfschema	dbug	man	source-downloads	win
INSTALL	extra	mysql-test	sql	zlib
README	include	mysys	sql-common	
VERSION	libbinlogevents	mysys_ssl	storage	
client	libbinlogstandalone	packaging	strings	

图 1.1

下表介绍了源码根目录中主要目录及文件的作用。

目录及文件	作用说明
BUILD	里面包含各个平台、各种编译器下进行编译的脚本
CMakeLists.txt	CMake 入口编译文件
client	客户端工具，所有的客户端工具都在这里，比如 mysql、mysqlbinlog、mysqladmin、mysqldump 等
cmake	为 CMake 编译服务的，这里定义了很多在 CMake 编译时使用的方法或变量
cmd-line-utils	一些小工具
config.h.cmake	用于生成编译时配置头文件的.cmake 文件
dbug	提供一些调试用的宏定义，可以很好地跟踪数据库执行到的执行函数、运行栈帧等信息，可以定位一些问题
extra	包含了用来做网络消息认证的 SSL 包，并提供了 comp_err、resolveip 等一些小工具
include	MySQL 代码包含的所有头文件，这里不包括存储引擎的头文件
libbinlogevents	MySQL 5.7 版本开始新增的、用于解析 Binlog 的 lib 服务
libmysql	用来创建嵌入式系统的 MySQL 客户端程序 API
libmysqld	MySQL 服务器的核心级 API 文件，也用来开发嵌入式系统
mysql-test	mysqld 的测试工具
mysys	MySQL 自己实现的一些常用的数据结构和算法，比如 array、list 和 hash，以及一些区分不同底层操作系统平台的函数封装，比如 my_file、my_fopen 等函数，这一类型的函数都以 my 开头
mysys_ssl	MySQL 中 SSL 相关的服务

续表

目录及文件	作用说明
plugin	包括一些系统内置的插件，比如 auth、password_validation 等，同时包含了可动态载入的插件，比如 fulltext、semisync 等
regex	一些关于正则表达式的算法实现
scripts	包含一些系统工具脚本，比如 mysql_install_db、mysqld_safe 及 mysql_multi 等
sql	MySQL 服务器主要代码，这里包含了 main 函数（main.cc），将会生成 mysqld 可执行文件
sql-common	存放部分服务器端和客户端都会用到的代码
storage	所有存储引擎的源代码都在这个目录中，文件夹名一般就是其存储引擎的名称，包括 innobase、myisam、blackhole、ndb 及 perfschema 等
strings	包含了很多关于字符串处理的函数，比如 strmov、strappend 及 my_atof 等函数
support-files	my.cnf 示例配置文件及编译所需的一些工具
unittest	单元测试文件目录
vio	虚拟网络 IO 处理系统，是对不同平台或不同协议的网络通信 API 的封装
win	在 windows 平台编译所需的文件和一些说明
zlib	zlib 压缩算法库（GNU）

建议大家看看每个目录中的文件，可以适当地望文生义，熟悉一下里面的内容及有关文件的存放位置，MySQL 的源代码大部分是按照功能而不是子系统来组织的，所以，可能在看代码的过程中发现特别乱，但其实熟悉并习惯了它的组织结构就很清晰了。

Linux 下的编译

在 Linux 下的编译与调试，通常不用 GUI 工具，操作起来更简单直接，编译过程中，依赖一些包及一些工具，需要先准备好，主要包括如下五个。

- cmake-2.8.12.2，在 Centos 上面，可以通过 yum install cmake 来安装，如果安装源没有，就自行下载源码来安装，过程比较简单。
- bison-2.4.1，同样可以使用 yum install bison 来安装，如果找不到的话，可以自行到 <http://gnuwin32.sourceforge.net/packages/bison.htm> 下载编译安装。
- libaio-devel，在 Centos 下面，可以直接通过命令 yum install libaio-devel.x86_64 安装。

- boost_1_59_0.tar.gz, 从 MySQL 官网就可以下载, 和 MySQL 源码放在一起。
- 如果缺少其他一些包的话, 一般是一些基础包, 可以相应下载安装即可。

然后, 将 mysql-5.7.16.tar.gz 解压, 解压之后可以看到其源码组织结构, 如图 1.1 所示。

可以从图 1.1 直观地看到, 在源码目录下有一个名为 CMakeLists.txt 的文件, 这个文件就是用来通过 CMake 配置编译环境的入口文件, 在每一个需要被编译的子目录中都有这么一个文件, 通过它们就可以在 Linux 下生成相应的 Makefile 文件了。

在编译之前, 最好是在源码目录下创建一个单独的目录, 用来存放编译中间文件等, 这样不会污染源码目录, 在想要清除环境的时候, 直接将这个编译目录删除即可。比如将这个目录叫作 debug, 则需要执行如下的命令。

```
cd mysql-5.7.16
mkdir debug
cd debug
```

再将下载好的文件 boost_1_59_0.tar.gz 放到源码目录中, 以便生成编译环境时使用。准备好之后, 通过下面的 CMake 命令来生成编译环境。

```
cmake .. -DBUILD_CONFIG=mysql_release \
-DINSTALL_LAYOUT=STANDALONE \
-DCMAKE_BUILD_TYPE=RelWithDebInfo \
-DENABLE_DTRACE=OFF \
-DWITH_EMBEDDED_SERVER=OFF \
-DWITH_INNODB_MEMCACHED=ON \
-DWITH_SSL=bundled \
-DWITH_ZLIB=system \
-DWITH_PAM=ON \
-DCMAKE_INSTALL_PREFIX=/var/mysql/ \
-DINSTALL_PLUGINDIR="/var/mysql/lib/plugin" \
-DDEFAULT_CHARSET=utf8 \
-DDEFAULT_COLLATION=utf8_general_ci \
-DWITH_EDITLINE=bundled \
-DFEATURE_SET=community \
-DCOMPILATION_COMMENT="MySQL Server (GPL)" \
-DWITH_DEBUG=OFF \
-DWITH_BOOST=..
```

上面的命令中, 有一点需要注意, 因为当前所在的相对目录为 mysql-5.7.16/debug, 而源码目录为 mysql-5.7.16, 所以 CMake 命令中指定源码目录为 “..”, 不然 CMake 找不到路径。

上面命令中用到了很多选项, 下面选择重要的来简单介绍一下。这些参数有的也是 CMake 的参数, 有兴趣学习 CMake 的同学, 可以专门找相应的资料来了解。

参数名	解释说明
CMAKE_BUILD_TYPE	这是编译选项，用来指定编译的是 RELEASE 版本还是 DEBUG 版本，或者是 RelWithDebInfo 版本的
CMAKE_INSTALL_PREFIX	这是用来指定安装路径的，指定的目录为 MySQL 安装之后的根目录
ENABLED_PROFILING	这个选项用来指定是否打开 profile 开关，一般想要查看 MySQL 执行信息时可以打开它
MYSQL_DATADIR	这个选项指定了在编译 INSTALL 工程时，生成的默认数据库路径，即在没有指定启动参数--defaults-file 时的默认数据库路径
OPTIMIZER_TRACE	用来指明是否打开优化器 TRACE 模块
WITH_ARCHIVE_STORAGE_ENGINE	表示是否支持 Archive 存储引擎
WITH_INNOBASE_STORAGE_ENGINE	表示是否支持 InnoDB 存储引擎
WITH_BLACKHOLE_STORAGE_ENGINE	表示是否支持 BLACKHOLE 存储引擎
WITH_EXAMPLE_STORAGE_ENGINE	支持 Example 存储引擎，这个存储引擎是一个示例，开发人员可以参考这个存储引擎来创建自己的存储引擎
WITH_FEDERATED_STORAGE_ENGINE	表示是否支持 Federated 存储引擎
WITH_INNODB_MEMCACHED	表示是否支持 INNODB 的 MEMCACHED
WITH_PARTITION_STORAGE_ENGINE	表示是否支持 partition 存储引擎
WITH_PERFSCHEMA_STORAGE_ENGINE	表示是否支持信息模式存储引擎

上面这些参数，如果有特别需求，可以在 CMake 命令中通过在前面加上-D 来指定相应的值，选择是否启用。

执行这个命令进行 CMake 的过程，就是配置编译环境的过程，执行完成之后，可以看到如图 1.2 所示的输出结果。

如果执行完之后，看到下面两行信息，则说明配置已经完成，可以做真正编译的工作了。

```
-- Configuring done
-- Generating done
```

此时可以发现，在当前目录下出现了 Makefile 文件，这就是上面 CMake 配置时生成的，有了 Makefile，就可以通过 make 来编译 MySQL 源码了。

```
-- Searching 16 bit integer
-- Using unsigned short
-- Check if the system is big endian - little endian
-- Gtest was not found. gtest-based unit tests will be disabled. You can run cmake . -DENABLE_DOWNLOADS=
1 to automatically download and build required components from source.
-- If you are inside a firewall, you may need to use an https proxy: export https_proxy=http://example.com:80
-- Performing Test HAVE_UNUSED_TYPEDEFS
-- Performing Test HAVE_UNUSED_TYPEDEFS - Failed
-- Performing Test HAVE_MISLEADING_INDENTATION
-- Performing Test HAVE_MISLEADING_INDENTATION - Failed
-- INSTALL mysqlclient.pc lib/pkgconfig
-- Skipping deb packaging on unsupported platform .
-- CMAKE_BUILD_TYPE: RelWithDebInfo
-- COMPILER_DEFINITIONS: _GNU_SOURCE; FILE_OFFSET_BITS=64;HAVE_CONFIG_H;HAVE_LIBEVENT1
-- CMAKE_C_FLAGS: -fPIC -Wall -Wextra -Wformat-security -Wvla -Wwrite-strings -Wdeclaration-after-statement
-- CMAKE_CXX_FLAGS: -fPIC -Wall -Wextra -Wformat-security -Wvla -Woverloaded-virtual -Wno-unused-parameter
-- CMAKE_C_FLAGS_RELWITHDEBINFO: -O3 -g -fabi-version=2 -fno-omit-frame-pointer -fno-strict-aliasing -DDEBUG_0
FF
-- CMAKE_CXX_FLAGS_RELWITHDEBINFO: -O3 -g -fabi-version=2 -fno-omit-frame-pointer -fno-strict-aliasing -DDEBUG_0
OFF
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zhufeng/mysql-5.7.16/debug
```

图 1.2

编译时，直接在 debug 下执行如下命令。

 make -j 24

此时，编译就开始了，后面加了参数-j 24，是想让编译过程快一些，因为编译机器比较好，所以设置了使用 24 个线程来编译。可想而知，非常快就完成了。

看到如图 1.3 所示的一些信息，说明已经编译完成。

```
[100%] Built target sql
Scanning dependencies of target pfs_connect_attr-t
Scanning dependencies of target mysqld
[100%] Building CXX object sql/CMakeFiles/mysqld.dir/main.cc.o
Linking CXX executable mysqld
[100%] [100%] [100%] Building CXX object storage/perfschema/unittest/CMakeFiles/pfs_connect_attr-t.dir/_
/_/_/sql/sql_builtin.cc.o
Building C object storage/perfschema/unittest/CMakeFiles/pfs_connect_attr-t.dir/_/_/_/mysys/string.c.o
Building CXX object storage/perfschema/unittest/CMakeFiles/pfs_connect_attr-t.dir/pfs_connect_attr-t.cc.o
Linking CXX executable pfs_connect_attr-t
[100%] Built target mysqld
Scanning dependencies of target udf_example
[100%] Building CXX object sql/CMakeFiles/udf_example.dir/udf_example.cc.o
Linking CXX shared module udf_example.so
[100%] Built target udf_example
[100%] Built target pfs_connect_attr-t
```

图 1.3

当然，如果想要对源码进行调试，需要将上面的 CMAKE_BUILD_TYPE 参数修改为 DEBUG，或者将参数 WITH_DEBUG 修改为 yes，不然是没有调试信息的。

编译完成之后，还需要安装好，需要执行如下命令。

```
make install
```

因为在 CMake 执行时，指定的参数 CMAKE_INSTALL_PREFIX 为 var/mysql，所以安装之后，所有关于 MySQL 需要的内容都会安装在这个目录下面，所包含的内容如下。

```
drwxr-xr-x  2 root root  4096 Jan 10 12:16 bin
-rw-r--r--  1 root root 17987 Nov 28 21:32 COPYING
drwxr-xr-x  2 root root  4096 Jan 10 12:16 docs
drwxr-xr-x  3 root root  4096 Jan 10 12:16 include
drwxr-xr-x  4 root root  4096 Jan 10 12:16 lib
drwxr-xr-x  4 root root  4096 Jan 10 12:16 man
drwxr-xr-x 10 root root  4096 Jan 10 12:16 mysql-test
-rw-r--r--  1 root root  2478 Nov 28 21:32 README
drwxr-xr-x 28 root root  4096 Jan 10 12:16 share
drwxr-xr-x  2 root root  4096 Jan 10 12:16 support-files
```

上面列出来的这些，包含了所有 MySQL 编译生成的可执行文件，其中包括 mysql、mysqld、mysqldump、mysqlbinlog 等文件，这些文件在 bin 目录下，这个目录，就是 MySQL 在运行时，需要指定的 basedir，在下面的配置文件中会看到这个参数。

安装 MySQL 库

编译完成了，接下来就是要创建一个新的数据库，对于 MySQL 5.7，安装方法与之前的版本有所不同，MySQL 5.7 的安装变得简单了很多，可以直接使用 mysqld 来安装，不过先来准备一个配置文件，名字默认为 my.cnf，放到目录/var/mysql/data_3306 下面，内容如下。

```
[mysqld]
port=3306
datadir=/var/mysql/data_3306
log_error=/var/mysql/data_3306/error.log
basedir=/var/mysql/
```

再来创建数据库，执行命令如下。

```
/var/mysql/bin/mysqld --defaults-file=/var/mysql/data_3306/my.cnf --initialize
--user=mysql
```

执行之后，在目录/var/mysql/data_3306/中生成的目录及文件如下。

```

-rw-r----- 1 mysql mysql      56 Jan 10 12:22 auto.cnf
-rw-r----- 1 mysql mysql     802 Jan 10 12:22 error.log
-rw-r----- 1 mysql mysql     413 Jan 10 12:22 ib_buffer_pool
-rw-r----- 1 mysql mysql 12582912 Jan 10 12:22 ibdata1
-rw-r----- 1 mysql mysql 50331648 Jan 10 12:22 ib_logfile0
-rw-r----- 1 mysql mysql 50331648 Jan 10 12:22 ib_logfile1
-rw-r--r-- 1 root  root       109 Jan 10 12:24 my.cnf
drwxr-x--- 2 mysql mysql    4096 Jan 10 12:22 mysql
drwxr-x--- 2 mysql mysql    4096 Jan 10 12:22 performance_schema
drwxr-x--- 2 mysql mysql   12288 Jan 10 12:22 sys

```

默认的数据库都生成了，还看到了 MySQL 5.7 中新加入的 sys 库。同时，也生成了指定的 error.log 文件。

从上面的结构中可以看到，所有的库文件及目录都是通过文件组 mysql 及用户 mysql 创建的。

接下来就是启动 MySQL 了，通过上面创建的配置文件，启动命令如下。

```
/var/mysql/bin/mysqld --defaults-file=/var/mysql/data_3306/my.cnf --user=mysql
```

此时，可以通过日志文件看到，服务器已经成功启动，也可以通过 `ps -ef|grep mysqld` 来查看进程信息。

MySQL 5.7 权限处理

启动完成之后，通过做一些简单的操作，可以测试一下，使用下面的命令。

```
mysql -uroot -h127.0.0.1 -P3306
```

系统直接报错，提示如下信息。

```
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

这个现象是正常的，在 MySQL 5.7 新版本中，调整了安全策略，新安装数据库之后，默认情况下 root 账号不像 5.6 一样没有密码，在 5.7 版本中，数据库启动时为 root 账号随机生成一个密码，存储在了 log_error 文件中，生成信息如下所示。

```
2017-01-10T04:22:05.200129Z 1 [Note] A temporary password is generated for
root@localhost: JqJhi1g,?oOr
```

这里可以看到，可以用来临时登录的用户是 root@localhost，而密码是 “JqJhi1g,?oOr”，这个密码很不规则，很复杂，安全性是比较高的，但只是用来临时登录的。在登录之后，MySQL 5.7 要求必须要修改密码，不然做不了其他任何事情，如图 1.4 所示。

```
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.16-debug

Copyright (c) 2009-2015 Percona LLC and/or its affiliates
Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql> show variables like "%sock%";
ERROR 1820 (HY000): You must reset your password using ALTER USER statement before
executing this statement.
```

图 1.4


从这里可以看到，MySQL 提醒，必须要通过 ALTER USER 语句修改密码，然后才可以正常使用，所以通过下面的语句，先修改了 root 账号。

```
 alter user 'root'@'localhost' identified by 'new password';
```

这样，从当前会话退出去之后，就可以使用这个密码来登录了。很明显，这样的处理，使得自动化程序变得比较麻烦。

所以，MySQL 5.7 提供了一种兼容方式，在初始化数据库时，可以通过指定参数 `--initialize-insecure` 来初始化，这样一来，其行为就与 MySQL 5.6 及之前版本是一样的了。这种方法给自动化运维带来了方便，不过背离了 MySQL 5.7 这种处理的初衷，所以需要想清楚这个可能造成的不安全因素。

这点从日志文件 `/var/mysql/data_3306/error.log` 中就可以看出来，内容如下。

```
 2017-01-10T04:45:56.249101Z 1 [Warning] root@localhost is created with an empty
password ! Please consider switching off the --initialize-insecure option.
```

2 MySQL 启动过程

一些应用程序的开发人员、DBA 及数据库爱好者想要深入了解 MySQL 内部实现方式，阅读其源代码是一个很好的途径，但是很多人反映说不知道如何下手，不知道如何找到一个正确的切入点，那么本章就会给出答案。

想要比较快速地了解 MySQL 源码，必须要让自己的学习步骤有一个比较清晰的脉络。它作为一个服务器，肯定会在启动之后一直处于运行状态，并且肯定有专门的线程一直监听客户端的连接及操作，这是一个基本的架构。那么，阅读代码时就首先根据这个脉络，有目的地去寻找，看看具体是如何实现的，当你找到它们的实现方法之后，也许会豁然开朗。

那么，首先看一下它的启动过程，看看它究竟干了些什么事情。mysqld 服务器是 C++ 语言生成的可执行文件，main 函数是其总的入口函数，程序从这里开始，那我们也从这里开始。

入口函数在 sql/main.cc 文件中。找到之后发现，这个文件中只有这一个函数，它又调用了 mysql_main，从这个函数开始到结束，就完成了 mysql 的启动操作。

```
/*  
    main() for mysqld.  
    Calls mysql_main() entry point exported by sql library.  
*/  
extern int mysql_main(int argc, char **argv);  
  
int main(int argc, char **argv)  
{
```

```

    return mysqld_main(argc, argv);
}

```

可以看出，所有操作都在 `mysqld_main` 中完成。下面这段代码，就是对 MySQL 入口函数 `mysqld_main` 做了精减，并保留了重要操作之后的样子，大概可以知道，在做完下面这些操作之后，MySQL 服务器就启动成功了。下面就这些代码，分别在注释处解释了它们的作用。

```

int mysqld_main(int argc, char **argv)
{
    my_progname= argv[0];
    orig_argc= argc;
    orig_argv= argv;

    /* 处理配置文件及启动参数等 */
    if (load_defaults(MYSQL_CONFIG_NAME, load_default_groups, &argc, &argv))
        return 1;
    sys_var_init();
    /* 继续处理参数变量 */
    ho_error= handle_early_options();
    mysql_audit_initialize();

    /* 日志系统初始化 */
    logger.init_base();

    /* 初始化很多系统内部变量 */
    if (init_common_variables())
        unireg_abort(1);          // Will do exit

    /* 信号系统初始化 */
    my_init_signals();

    /* 核心模块启动，包括存储引擎等 */
    if (init_server_components())
        unireg_abort(1);
    if (init_ssl())
        unireg_abort(1);

    /* 终端重定向处理 */
    reopen_fstreams();

    /* 网络系统初始化 */

```

```

network_init();
start_signal_handler();          // Creates pidfile
if (!opt_noacl)
    (void) grant_init();

/* 状态变量初始化 */
init_status_vars();

/* Binlog相关检查初始化 */
check_binlog_cache_size(NULL);
check_binlog_stmt_cache_size(NULL);
if (!opt_bootstrap)
{
    set_slave_skip_errors(&opt_slave_skip_errors);
    if (server_id != 0)
        init_slave(); /* Ignoring errors while configuring replication. */
}

create_shutdown_thread();
start_handle_manager();

/* 服务监听线程创建 */
handle_connections_sockets();
/* Wait until cleanup is done
从这里开始，服务器启动线程一直等待，直到shutdown后，继续向下执行*/
mysql_mutex_lock(&LOCK_thread_count);
while (!ready_to_exit)
    mysql_cond_wait(&COND_thread_count, &LOCK_thread_count);
mysql_mutex_unlock(&LOCK_thread_count);
clean_up(1);
mysqld_exit(0);
}

```

在 load_defaults 中，首先会构造默认搜索配置文件的路径，在 windows 下包括 C:\Windows\System32、C:\Windows\、mysqld 所在目录及 MySQL 的安装目录等，在 Linux 下包括/etc/及/etc/mysql 等，然后解析 mysqld 的执行命令，一般情况会在后面设置--defaults-file 参数，但也可以设置其他的参数。如果发现这里设置了--defaults-file，就确定了启动参数文件为这个参数指定的文件；如果没有这个参数，系统就会从上面已经构造的几个系统目录中找名为 my.cnf 及 my.ini 的文件，如果最终没有找到，则系统退出。

当确定了配置文件之后，系统通过函数 search_default_file_with_ext 打开并解析每一行内容，因为配置文件支持分组，所以它同时会确定当前解析的参数属于哪个组。分组一般包括

mysqld、server、mysql5.6（取决于具体版本）等，初始值是由 load_default_groups 指定的，同时跳过所有以 # 开头的注释行。对于解析到的每一个有效参数设置，都会被标准化并且缓存起来，标准化是在前面加上--，比如在文件中是 datadir=/data/mysql/mysqldata_slave，那么标准化后就是--datadir=/data/mysql/mysqldata_slave，这可以联想到在启动 mysqld 的时候设置的参数--defaults-file，是一个道理，最终都在里面转化为启动参数。

每个参数都会被缓存到内存中，这个缓存操作是由函数 handle_default_option 来完成的，这里用到如下一个结构。

```
struct handle_option_ctx
{
    MEM_ROOT *alloc;
    DYNAMIC_ARRAY *args;
    TYPELIB *group;
};
```

结构中的 group 用来指明 args 数组中存储的所有参数共同属于哪一组，它是预先设定好的，alloc 是它们的内存空间。从这个函数可以看出，它只处理当前 handle_option_ctx->group 中含有的组名（这次只是服务器），其他的都被暂时忽略了，这里说的是暂时忽略，因为系统启动时，后面还会再处理其他类型的参数。

那么，上面这个操作是将 load_default_groups 指定的组的参数缓存起来，然后通过 my_load_defaults 的参数 argv 传到上层栈帧去。此时，load_defaults 所做的操作就完成了，它的作用就是找到配置文件并从中解析出服务器对应的参数。

接下来的函数是 sys_var_init，这里是将所有的系统变量加入到哈希表 system_variable_hash 中。那么，这些系统变量是哪里来的呢？这里可以看到，它用的是 all_sys_vars，这是一个单链表，它包含了 sys_vars.cc 文件中指定的所有类似 Sys_pfs_enabled 的变量，这些变量都是全局变量，在没有执行 main 函数时就加入到链表中了，C++ 会在 main 函数之前执行它们的（类 sys_var）构造函数，在构造函数中有将其加入到链表的代码。

接下来要做的是 handle_early_options，还是对配置参数的处理。但此处有些特殊，因为在 MySQL 启动时，有些模块必须很早就要执行，而它会用到一些参数，所以就将参数按照使用顺序分为了 PARSE_EARLY 和 PARSE_NORMAL 两部分，区分这两个不同属性，要看在创建一个参数时通过类（sys_var）成员 m_parse_flag 所指定的值是什么，比如参数 max_connections，如下。

```
static Sys_var_ulong Sys_max_connections(
    ...// 中间部分省略
    sys_var::PARSE_EARLY);
```

handle_early_options 里调用了一个很重要的函数 handle_option，如下。



```
handle_options(int *argc, char ***argv, const struct my_option
               *longopts, my_get_one_option get_one_option)
```

它用来设置某一批变量，设置的目标地址是 longopts，这里面存储的是所有 PARSE_EARLY 类型的变量地址，源变量信息存储在 argv 数组中，个数通过 argc 来表示，这里还是上面 load_defaults 函数生成的服务器变量信息，每一个值都还是用类似 --datadir=/data/mysql/mysqldata_slave 这样的形式来表示的。get_one_option 是一个函数指针，用来解析变量类型并且设置一些特定类型的变量值，这里当目标变量与源变量信息有交集时才会设置，其他没有找到的变量直接使用其默认值。

接下来是 logger.init_base。logger 是 MySQL 用来记录系统运行情况的日志系统，默认情况下，系统会将日志信息写入到文件中，但可以通过设置将其记录到表中。

接下来的一个接口 init_common_variables 是用来初始化一些全局变量的，包括对配置文件的进一步处理，从这里可以看到，MySQL 大量使用了全局变量，让人眼花缭乱。下面先看看这个函数精简后的样子，基本都是对一些系统的状态参数或者用户自定义变量的初始化及调整。



```
int init_common_variables()
{
    /* 初始化默认存储引擎，在5.5版本之后，默认存储引擎都是InnoDB */
    default_storage_engine= const_cast<char *>("InnoDB");
    default_tmp_storage_engine= default_storage_engine;

    /* 初始化权限检查插件 */
    init_default_auth_plugin();

    /* 初始化所有状态变量，这些变量就是我们一般使用的命令
    "show status like "%variable_name%"所展示出来的变量值 */
    if (add_status_vars(status_vars))
        return 1; // an error was already reported

    if (get_options(&remaining_argc, &remaining_argv))
        return 1;
    set_server_version();

    /* Calculate and update default value for thread_cache_size. */
    if ((default_value= 8 + max_connections / 100) > 100)
        default_value= 100;
```

```

var= intern_find_sys_var(String_with_len("thread_cache_size"));
var->update_default(default_value);

/* Calculate and update default value for host_cache_size. */
/* 这里可以看到一个很熟悉的参数: max_connections,
   且参数host_cache_size、thread_cache_size的值,
   是根据max_connections的值来调整的 */
if ((default_value= 128 + max_connections) > 628 &&
    (default_value= 628 + ((max_connections - 500) / 20)) > 2000)
    default_value= 2000;
var= intern_find_sys_var(String_with_len("host_cache_size"));
var->update_default(default_value);

/* Fix thread_cache_size. */
if (!thread_cache_size_specified &&
    (max_blocked_pthreads= 8 + max_connections / 100) > 100)
    max_blocked_pthreads= 100;

/* Fix host_cache_size. */
if (!host_cache_size_specified &&
    (host_cache_size= 128 + max_connections) > 628 &&
    (host_cache_size= 628 + ((max_connections - 500) / 20)) > 2000)
    host_cache_size= 2000;

/* Read error messages from file, 所有MySQL给客户端返回
   的错误信息, 都是通过这个操作从errmsgmessage信息中读取出来的*/
if (init_errmessage())
    return 1;
/* 初始化词法分析系统 */
lex_init();
/* 初始化默认的字符集 */
global_system_variables.collation_server= default_charset_info;
global_system_variables.collation_database= default_charset_info;

/* 处理慢查询日志文件的存储方式, 需要根据日志存储方式来确认
   是通过文件还是表来存储 */
if (opt_slow_log && opt_slow_logname && !(log_output_options & LOG_FILE)
    && !(log_output_options & LOG_NONE))
    sql_print_warning("Although a path was specified for the "
        "--slow-query-log-file option, log tables are used. "
        "To enable logging to files use the --log-output=file option.");

```

```

/* 确定日志文件的名字 */
FIX_LOG_VAR(opt_logname,
             make_default_log_name(logname_path, ".log"));
FIX_LOG_VAR(opt_slow_logname,
             make_default_log_name(slow_logname_path, "-slow.log"));
return false;
}

```

首先可以看到, 设置默认存储引擎 `default_storage_engine` 也是在这里做的, 之后初始化了权限验证插件。

接着, 函数 `add_status_vars` 是将所有的状态变量存储到 `all_status_vars` 中, 这也是我们经常使用的 `show status` 命令所对应的信息。

接下来, 再次调用 `get_option` 函数, 这里还是带着前面 `load_defaults` 所得到的参数信息, 用这些参数继续初始化系统变量。这次初始化的目标是 `mysqld.cc` 文件中的结构体数组 `my_long_options`, 再加上上面介绍的 `sys_vars.cc` 中的所有变量, 不过这次使用的是 `PARSE_NORMAL` 类型的, 因为另一种已经被设置过了, 设置过程和上面是一样的。现在是不是感觉 MySQL 中的参数设置很乱? 设置次数太多, 并且有很多类型的参数。其实, MySQL 对参数是分类的, 在 `sys_vars.cc` 中可以在客户端通过 `show variables` 命令查看, 而在 `my_long_options` 中可以被指定为命令行启动参数, 如果变量同时出现在这两类中, 说明它同时具备这两种特性。

再往下, 还是对一些零散全局变量的初始化操作。

那么, 针对函数 `init_common_variables` 的功能, 上面通过注释的方式在代码中已经清楚地讲述过了, 可以看出, 正如其名字所表达的, 都是对一些系统变量的处理, 这里不做过多的叙述了。

再回到上面的函数 `mysqld_main` 中, 继续下一个核心函数 `init_server_components`。

在核心函数 `init_server_components` 中, 将初始化一些系统模块, 包括 `mdl` (元数据锁)、表定义缓存、查询结果集缓存、创建错误日志文件。如果配置了 `Binlog`, 则初始化并打开 `Binlog` 系统, 然后初始化存储引擎, 它会将所有系统支持且经过配置的存储引擎初始化, 具体的 `InnoDB` 存储引擎的初始化在第 5 章会有专门介绍。

继续跟踪下去, 下面是 `network_init`。这个函数的主要作用是创建服务器的 `socket`, 并绑定端口, 然后调用 `listen` 函数将当前套接字转换为被动套接字。这里并没有真正创建监听线程, 不必着急, 再往下看。

接着是函数 `reopen_fstreams`, 这就是为什么不管在 Windows 还是 Linux 下, 系统内部的 `printf` 都没有显示在终端上, 而是被记录在 `log` 文件中的原因, 因为这里将输出重定位了。所有的输出都被重定位到文件中, 如果为了方便调试, 可以稍微修改一下这里的代码, 就可以让信息重新显示到终端。

接下来,调用函数 `grant_init`,它是在变量 `opt_noacl` 为0的时候才被执行的。而看一下 `opt_noacl` 这个变量在什么地方初始化就会发现,原来它是通过参数 `skip-grant-tables` 设置的,如果设置这个参数为真,那么任何操作都不再受权限限制。

再下来就是 `init_slave` 函数,它是初始化复制的。如果之前没有配置过向其他主库的复制操作,或者设置了参数 `--skip_slave_start`,这里就会在初始化之后直接返回;而如果之前已经开启过复制,同时又没有设置这个参数,这里就会同时开始复制操作。

最后,MySQL 做的另外一个重要的操作就是调用函数 `handle_connections_sockets`。这个函数与创建用户线程有关系,所以会在第3章“用户连接线程创建”一节中做详细介绍。

到这里,MySQL 的启动就完成了,从上面步骤中可以看到,涉及的内容比较多,特别是存储引擎方面的东西,关于存储引擎,后面会有更多章节来讲述。

这是 MySQL 启动过程的一个大概轮廓,一开始没必要了解太深入,只要对上面提到的内容保持大概了解,并且在说起来时能表现出一种“见过”的状态就够了。因为这是刚开始,所以需要不断深入,这里只需要了解一个大概即可。

3 连接的生命与使命

用户连接线程创建

第2章中提到，在服务器启动的过程中，只是创建了套接字且绑定了端口，但还没有对其进行监听，也就是还没有形成服务器监听模式，这些操作是在函数 `handle_connections_methods` 中完成，这里所做的工作就是等待一个用户的连接请求，并给它分配一个工作线程。

首先，在这个函数中会处理三种连接方式，分别是命名管道、套接字及共享内存。一般情况下，都是用套接字的连接方式，其他连接方式只有在一定条件下才能使用，所以这里只看关于套接字的。我们会发现在一个大的循环里面有 `select` 或者 `poll`（网络编程），然后通过 `mysql_socket_accept` 生成一个新的套接字，这个就是真正针对新连接的连接套接字，这个套接字才是用来监听连接请求的，而服务器的连接监听套接字会在创建新连接线程（`create_new_thread`）之后，返回循环中继续监控有没有新的连接进来。

可以通过如下代码来看一下，针对套接字方式的连接请求的处理，MySQL 是如何做的。

```
void handle_connections_sockets()
{
    /* local variables ... */
    while (!abort_loop)
    {
        /* 首先看到的就是我们所熟悉的服务器监听操作，这里有两种方式，
```


一种是POLL，另一种就是SELECT方式了，这属于网络编程的范畴，
可以找一些书籍详细了解一下 */

```
#ifdef HAVE_POLL
    retval= poll(fds, socket_count, -1);
#else
    readFDs=clientFDs;
    retval= select((int) max_used_connection,&readFDs,0,0,0);
#endif

    if (retval < 0)
    {
        /* exception handle codes ... */
        continue;
    }

    /* Is this a new connection request ?
    如果上面的监听有消息返回了，说明有新的连接在请求，这里针对
    不同监听方式做不同处理，分别取出其中对应的sock，等待进一步
    网络交互 */
#ifdef HAVE_POLL
    for (int i= 0; i < socket_count; ++i)
    {
        if (fds[i].revents & POLLIN)
        {
            sock= pfs_fds[i];
            flags= 0;
            break;
        }
    }
#else // HAVE_POLL
    if (FD_ISSET(mysql_socket_getfd(base_ip_sock), &readFDs)) {
        sock = base_ip_sock;
        flags= ip_flags;
    }
    else
    if (FD_ISSET(mysql_socket_getfd(extra_ip_sock), &readFDs)) {
        sock = extra_ip_sock;
        flags= extra_ip_flags;
    }
#endif // HAVE_POLL
```

```

/* 下面是熟悉的网络编程中的accept, 如果成功了, 则会产生一个新的
   套接字出来, 这样说明握手成功, 连接建立成功, 那么这个新的套接
   字就会与这个连接所对应, 产生的新套接字为new_sock */
for (uint retry=0; retry < MAX_ACCEPT_RETRY; retry++)
{
    size_socket length= sizeof(struct sockaddr_storage);
    new_sock= mysql_socket_accept(key_socket_client_connection, sock,
                                   (struct sockaddr *)&cAddr, &length);
    if (mysql_socket_getfd(new_sock) != INVALID_SOCKET ||
        (socket_errno != SOCKET_EINTR && socket_errno != SOCKET_EAGAIN))
        break;
}

/* 为新建连接创建线程对象thd, 所有关于这个连接的信息,
   都用这个对象来保存并处理, 了解一些代码的同学应该对THD
   结构很清楚了, 从这里可以看出, 在连接还没有创建完成的时候,
   这个THD就已经创建了 */
if (!(thd= new THD))
{
    /* 异常处理, 如果创建失败就关掉套接字, 主监听线程继续它的工作 */
    (void) mysql_socket_shutdown(new_sock, SHUT_RDWR);
    (void) mysql_socket_close(new_sock);
    statistic_increment(connection_errors_internal, &LOCK_status);
    continue;
}

/* 这里是为当前连接创建一个vio, 主要是处理网络数据传输的,
   这个对象也会一直陪伴着这个线程对象thd, new_sock是为这个连接
   产生的新套接字, vio也是建立在这个套接字基础之上的 */
vio_tmp= mysql_socket_vio_new(new_sock, vio_type, vio_flags);

/* 为当前连接初始化网络数据传输相关的参数, 并取得连接文件句柄 */
if (!vio_tmp || my_net_init(&thd->net, vio_tmp))
{
    /* exception handle codes */
    continue;
}

/* other codes ... */
/* 对于当前连接, 这是最重要的一个操作, 是真正创建连接的操作。
   下面会继续解释这个函数 */
create_new_thread(thd);

```

```

    }
    DEBUG_VOID_RETURN;
}

```

从上面代码中可以看到，创建一个新的连接，最终是通过一个核心函数 `create_new_thread` 实现的，创建的就是前面所说的工作线程，新的套接字是与这个工作线程绑定的。下面是这个函数的伪代码。

```

static void create_new_thread(THD *thd) {
    NET *net=&thd->net;
    /* 判断当前连接数是不是超过了系统配置允许的最大值，如果是就断开连接 */
    if (connection_count >= max_connections + 1 || abort_loop) {
        close_connection(thd, ER_CON_COUNT_ERROR, 1);
        delete thd;
    }
    ++connection_count;
    /* 将新连接加入到thread_scheduler的连接队列中 */
    thread_scheduler.add_connection(thd);
}

```

这里有我们很熟悉的 `max_connections`，没错，就是数据库配置参数中的 `max_connections`。`connection_count` 表示当前系统中的连接个数，当这个值大于 `max_connections` 时，服务器就拒绝连接，直接报错返回。

需要注意，这里还没有真正地创建连接，而只是判断连接数目而已，真正的连接是上面代码的最后一行，其调用的真正函数如下。

```

void create_thread_to_handle_connection(THD *thd) {
    /* 看当前工作线程缓存 (thread_cache) 中是否有空余的线程。
       如果有的话则唤醒一个线程来用，而不再新创建一个线程，因为创建
       一个新的线程，也是有成本的 */
    if (cached_thread_count > wake_thread) {
        thread_cache.append(thd);
        pthread_cond_signal(&COND_thread_cache);
    } else {
        /* 将当前thd保存到threads列表中。这个列表，就是我们
           很熟悉的“show processlist;”信息的来源 */
        threads.append(thd);
        /* 创建一个新的线程，这个线程handle_one_connection
           与这个连接唯一对应（如果没有开启连接池模式的话） */
        pthread_create(&thd->real_id,&connection_attr,

```

```

        handle_one_connection,(void*) thd)));
    }
}

```

这里是对连接池的处理。因为在 MySQL 中，用户连接退出后，在服务器中其实并没有真正地销毁工作线程，而是将它缓存起来，如果以后有新的连接请求，系统会将已经缓存起来的连接直接交给这个用户使用，这样可以在一定程度上提高性能。如果没有找到空余的线程，就重新创建一个线程，pthread_create 的第三个参数就是这个线程对应的线程处理函数，从上面的代码可以看到，它是 handle_one_connection。

```

pthread_handler_t handle_one_connection(void *arg) {
    /* 初始化线程预处理操作 */
    thread_scheduler.init_new_connection_thread();
    /* 载入一些Session级变量 */
    setup_connection_thread_globals(thd);
    /* 第一层死循环 */
    for (;;) {
        /* 初始化LEX词法解析器 */
        lex_start(thd);
        /* 进行连接身份验证 */
        login_connection(thd);
        /* 初始化线程，准备执行语句 */
        prepare_new_connection_state(thd);
        /* 看得出，这是一个死循环，只要alive，就会一直运行下去。
           当alive为false时，说明这个连接就断了，断了之后，会执行
           下面的函数end_connection，即断掉一个连接的操作 */
        while(alive)
            do_command(thd); /* 处理命令，核心操作 */
        /* 操作结束，关闭连接，丢入线程池 */
        end_connection(thd);
    }
}

```

这里有两层循环。第一层是 for 循环，这是为了处理上面所说的连接池线程重用的，当一个连接断开后，系统会将这个线程放入到连接池中，但这个线程被阻塞到这个循环内部，当这个线程被重用之后，它会从 lex_start 开始重新执行并登录，登录成功后，将继续做命令的处理工作。第二层循环就是 while(alive)，这个循环是用来处理一条条命令的，每当处理完一条命令，就会循环一次。这里主要看一下 do_command 这个函数里面做了什么操作，如下。

```

bool do_command(THD *thd) {
    NET *net= &thd->net;
    packet_length = my_net_read(net);
    packet = (char*) net->read_pos;
    /* 解析客户端传过来的命令类型 */
    command = (enum enum_server_command) (uchar) packet[0];
    dispatch_command(command, thd, packet+1, (uint) (packet_length-1));
}

```

从这里可以很清楚地看到，原来每一次循环都会从网络读取数据，读取函数为 `my_net_read`，读取的内容被存储在 `packet` 中，这其实是一个字符数组，存储的是这个连接线程中客户端读取的 MySQL 消息，而 `packet_length` 就是用来表示当前命令在 `packet` 中的长度的。这样也就可以知道，当一个连接没有任何请求，或者一个连接新创建的时候，线程就是阻塞在这里的，只有当请求到来时，这个线程才会继续向前执行。

到此为止，一个用户线程从请求，到一步步的创建，最后到阻塞在网络读取的位置，说明用户线程已经创建完成。但是很明显的一个问题就是，每一个会话都对应一个线程，而操作系统的资源是有限的，当会话数达到一定数目时，MySQL 的性能会急剧下降，所以，目前在很多 MySQL 应用中，一般都限制了 `max_connections` 值来防止会话太多的情况。

MySQL 处理请求

上一节讲述了一个线程的创建过程。当线程创建完成后，它被阻塞在网络读取上，当一个请求到达时，线程会从读取接口返回，并得到请求命令的信息。至此，终于要做任务解析了。

图 3.1 说明了从客户端向服务器请求的消息传输方式及过程。

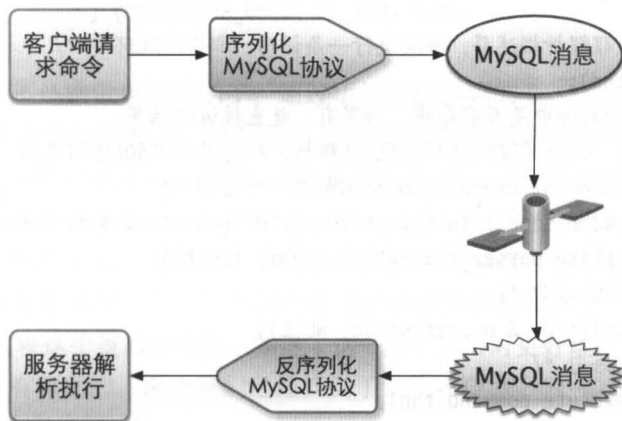


图 3.1

这里从 packet 中可以读取到请求信息，它的第一个字节表示了这个请求的类型，在源码中，可以通过枚举 enum_server_command 看到 MySQL 支持哪些操作类型。然后，函数 dispatch_command 所做的工作就是根据请求类型做相应的事情，可以具体看一下它的情况，如下。

```
bool dispatch_command(enum enum_server_command command,
                      THD *thd, char* packet, uint packet_length) {
    NET *net = &thd->net;
    thd->command = command;
    /* 分门别类，不同命令类型，做不同处理 */
    switch (command) {
        case COM_INIT_DB: ...;
        case COM_TABLE_DUMP: ...;
        case COM_CHANGE_USER: ...;
        ...
        case COM_QUERY: //如果是Query
            /* 从网络数据包中读取Query并存入thd->query */
            alloc_query(thd, packet, packet_length);
            /* 送去解析 */
            mysql_parse(thd, thd->query, thd->query_length, &end_of_stmt);
    }
}
```

从上面代码中可以看到，通过一个非常大的 switch 语句，根据不同的类型做不同的处理操作。来看我们最熟悉的 COM_QUERY，看上去它是处理查询的，其实它是处理所有对数据库访问操作的，比如 dml、ddl 等 SQL 语句，这个请求的处理函数是 mysql_parse，如下。

```
void mysql_parse(THD *thd, const char *inBuf, uint length,
                 const char ** found_semicolon) {
    /* 初始化线程解析描述符，每次执行一条语句，都会清理环境 */
    lex_start(thd);
    /* 看query cache中是否有命中，如果有，就直接返回结果；
       如果没有，就会通过词法语法做SQL解析，知道是什么SQL语句之后，
       再通过函数mysql_execute_command做进一步处理 */
    if (query_cache_send_result_to_client(thd, (char*) inBuf, length) <= 0) {
        Parser_state parser_state(thd, inBuf, length);
        /* 解析SQL语句 */
        parse_sql(thd, & parser_state, NULL);
        /* 执行语句 */
        mysql_execute_command(thd);
    }
}
```

这里有一点值得注意，就是开始位置对查询缓存的访问。查询缓存中存放着最近一段时间内，出现频率最高的语句和它对应的查询结果。这个函数首先判断当前处理的语句在不在 Query Cache 中（对于 MySQL 的查询缓存，其实功能非常弱，因为只有当语句完全相同，包括空格及大小写也要完全相同的情况下才能命中，但这样的情况一般是很少出现的）。如果当前处理的语句存在于 Query Cache 中，则直接将其对应的结果集返回给用户；如果没有，则通过 `parse_sql` 对 SQL 语句进行分析，这主要是系统根据系统语法文件 `sql_yacc.yy` 进行语法分析，生成相应的语法树，然后通过函数 `mysql_execute_command` 来执行这条语句，如下。

```
int mysql_execute_command(THD *thd) {
    /* 解析过后的SQL语句的语法结构 */
    LEX *lex= thd->lex;
    /* 该语句要访问的表的列表 */
    TABLE_LIST *all_tables = lex->query_tables;
    switch (lex->sql_command) {
        ...
        case SQLCOM_INSERT:
            insert_precheck(thd, all_tables);
            mysql_insert(thd, all_tables, lex->field_list, lex->many_values,
                lex->update_list, lex->value_list, lex->duplicates, lex->ignore);
            break; ...
        case SQLCOM_SELECT:
            /* 检查用户对数据表的访问权限 */
            check_table_access(thd, lex->exchange ? SELECT_ACL |
                FILE_ACL : SELECT_ACL, all_tables, UINT_MAX, FALSE);
            /* 执行select语句 */
            execute_sqlcom_select(thd, all_tables);
            break;
    }
}
```

看到上面的代码是不是感觉更加明朗了？通过语法分析得到的 `lex->sql_command` 知道当前 SQL 语句是一个什么类型的操作，这里可以看到我们最熟悉的 `SQLCOM_INSERT`。没错，`SQLCOM_INSERT` 对应的就是插入语句，`SQLCOM_SELECT` 对应的就是查询语句，它们分别对应的操作函数为 `mysql_insert` 与 `execute_sqlcom_select`。通过各自对应的操作函数，就可以将对应的操作执行完成。至于这两个函数内部的实现情况，这里就不准备更深入地讲述了，这里的目标就是展示一下，一个用户从连接请求到执行操作是一个怎样的过程，在 MySQL 中是如何实现这个逻辑架构的，知道了这一点，其实对更好地理解 MySQL 运行原理及实现方式是非常有帮助的，因为我们已经找到了它的入口。

上面提到过语法分析，主的入口函数是 `parse_sql`，进而再调用 `MYSQLparse` 函数，而这个函数是通过一个宏直接指向了 `yyparse`，实际上这才是真正的语法分析函数，它在 `sql_yacc.cc`

中实现, 这个文件是通过上面提到的 `sql_yacc.yy` 生成的, 如果安装了 `bison`, 那么可以试试如何生成 `sql_yacc.cc` 文件。可以大致浏览一下这个生成文件的内容, 它包含了一条很长的 `switch` 语句, 每一种 `case` 都是一种语句解析状态。其实一般情况下, 在学习时都不需要看这个文件, 只需要关注 `.yy` 文件即可, 因为通过这个文件可以更简单明了地知道一条语句的语法情况, 举一个小例子, 如下。

```
insert:
    INSERT
    {
        LEX *lex= Lex;
        lex->sql_command= SQLCOM_INSERT;
        lex->duplicates= DUP_ERROR;
        mysql_init_select(lex);
    }
    insert_lock_option
    opt_ignore insert2
    {
        Select->set_lock_for_tables($3);
        Lex->current_select= &Lex->select_lex;
    }
    insert_field_spec opt_insert_update
    {}
;
```

上面的代码是从 `sql_yacc.yy` 文件中截取的一段关于 `insert` 语句的语法分支。最前面的 `insert` 表示 `insert` 语句的语法分支, 直到遇到分号才结束, 中间部分就是语句所要匹配的语法规项。大写 `INSERT` 指的就是 `insert` 字符串本身, 仅仅用来表示它是一条插入语句, 除此之外没有任何用处。这个词后面的大括号中包含的内容就是构造一个插入语句类型, 再向后的 `insert_lock_option`, 也是一个语法分析, 用来指定这个插入语句的锁模式, 带有 `option` 表示这个是可选项, 这点可以从其语法分支中看出, 如下。

```
insert_lock_option:
    {$$= TL_WRITE_CONCURRENT_INSERT;}
    | LOW_PRIORITY { $$= TL_WRITE_LOW_PRIORITY; }
    | DELAYED_SYM { $$= TL_WRITE_DELAYED; }
    | HIGH_PRIORITY { $$= TL_WRITE; }
;
```

为了节省篇幅, 这里省略了一部分, 从这个分支的第一个选项中可以看到, 它没有任何要匹配的内容, `$$` 表示当前语法分支的返回值, 这里的返回值为默认值 `TL_WRITE_CONCURRENT_INSERT`, “|” 表示它两边的内容是并列的, 要么选择前面, 要么选择后面, 上面可以选择

的有四种,除了默认情况还包括 LOW_PRIORITY、DELAYED_SYM、HIGH_PRIORITY,分别返回不同的值,这个值会被放到分析结果中,在执行时会用到。

继续看 insert 分支,后面是 opt_ignore,同样地,它也是一个分支,这里不详细说明。

再向后是一个 insert2 分支,如下。

```
insert2:
    INTO insert_table {}
    | insert_table {}
;
```

顾名思义,insert_table 这个语法分支表示的是表名或库名.表名。这个不用细看,前面一个有 INTO,后面一个没有,中间用“|”连接,说明插入语句原来还可以没有 INTO,不妨试试看。

再回到 insert 语法分支,接着是一个大括号,这里面出现了一个 \$3,它表示的是一个语法项的返回值,从 1 开始,一直向后数,大括号也算。从上面可以看到,语句 Select->set_lock_for_tables(\$3) 中的参数 \$3 指的正是 insert_lock_option,这个语法项在上面已经描述过了,\$3 表示的值就是它的返回值。

再接下来的两个语法项和前面的道理是一样的,分别指定了插入列及插入数据信息和 ON DUPLICATE 语句信息等。

总结

一个简单的例子就可以说这么多,如果想看其他语句的语法,可以继续深入研究。在我看来,MySQL 的语法文件写得特别复杂,也特别乱,可扩展性也特别差,在一般的数据库理论中,语句的执行要分为词法分析、语法分析、语义分析、查询优化、计划生成等步骤,但 MySQL 将这里面很多步骤都揉合到一起,完全没有层次划分,更为严重的是,它的存储过程、函数的执行,将计划生成放在了语法、语义分析的过程中,这一点个人认为不太好,因为这样一来很多方面受到局限,比如语句扩展、优化、性能等。

4

MySQL 表对象缓存

表结构的实现原理

在 MySQL 中，有很多类型的系统对象，包括表、视图、存储过程、存储函数等，但由于 MySQL 的插件式存储引擎及其他实现方面的特点，导致每一种对象的缓存方式都不同，或者说这些对象的缓存不是通过一种统一的方式来管理，而是都有自己的特点，并且缓存的内容也有很大的差异。下面先只叙述一下表对象的缓存方式。

表对象缓存，顾名思义，是将某个表对象的字典信息（定义内容）缓存到内存中，用来提高对表访问的效率。某个表被访问过一次之后，在服务器没有关闭且表定义没有被修改的条件下，访问这个表时，只需要从内存中找到这个已经缓存起来的对象并做相应操作即可，而不必再次从系统表中读取它的定义并解析，然后再做相应的操作。

当某个用户要查询某一个表的数据时，系统首先会找到这个表。上面已经提到过，因为 MySQL 实现了表的缓存，所以首先会从缓存中寻找这个表。表字典对象的缓存是通过 HASH 表来管理的，MySQL 系统中，专门有一个 HASH 表（源代码中的名字是 `table_def_cache`）用来存储组织表对象，通过表的名字（包括了模式名）来构造一个 HASH 键值（Key），用来从 HASH 表中搜索对象。

但是对于表对象的缓存，不只是简单地将一些表的定义通过 HASH 存储起来就算完成，如果这样的话，缓存可能没有任何意义，或者说局限性非常大。这样可能导致一个用户在表对象上做了任何标志或修改等都会影响到其他用户，这种影响是不可预期的。更重要的原因是，MySQL 是插件式的数据库，每一个用户得到表对象之后还需要将表实例化，这个实例化的

对象只有自己才能使用，所以并不是简单的所有用户都使用同一个缓存对象即可完成的。

它的缓存其实是用了一种“共享私有化缓存”。看上去这个说法是矛盾的，其实不然，它在缓存过程中用到一个叫 TABLE_SHARE 的结构体，这个结构体唯一对应 MySQL 中的一个表对象，这里是不区分任何存储引擎的，它实际上就是对具体一个表的定义的翻译或映射。

在打开一个表的时候，这个表首先是在 MySQL 的系统表中存储的。这里的系统表，分两个层次，一个层次是 MySQL 的 .frm 文件，这是共有的，与存储引擎没有关系，也属于一种系统表；另一层就要分不同的存储引擎，不同的存储引擎有自己的系统表，比如 InnoDB（关于这个话题在第6章中有专门介绍），所以这里所说的 MySQL 的系统表应该是一种统称。

打开表时，首先需要从系统表中将这个表的所有信息都读入到内存中，这些信息包括表名、模式名（在 MySQL 中等同于库名）、所有的列信息、列的默认值、表的字符集、对应的 .frm 文件的路径、所属的存储引擎（MySQL 中的表可以单独定义自己的存储引擎）、主键等，当然还有很多其他信息。所有这些信息读入内存中的时候，首先是通过结构体 TABLE_SHARE 来存储的，相当于这个结构体是一个表对象缓存的第一层。同时，从名字就可以看出，这个结构体是所有用户都可以共享的一个表对象，所以它是静态的，不允许修改的（内存中），从系统表中读取进来之后直到这个表从缓存中删除，中间不会做任何修改。

用户要访问一个表，只构造了 TABLE_SHARE 是远远不够的，而且这个结构体对象也不是直接给用户使用的对象。构造了这个结构体之后，首先需要将其缓存起来，因为这个结构体就是这里讨论的核心，它就是要缓存的对象，所以首先需要根据上面计算得到的 Key，将这个表对象缓存到 table_def_cache 中，这个缓存操作到这里就结束。

对应的精简代码版本如下。

```
TABLE_SHARE *get_table_share(THD *thd, TABLE_LIST *table_list,
                             const char *key, size_t key_length,
                             uint db_flags, int *error,
                             my_hash_value_type hash_value)
{
    TABLE_SHARE *share;
    int open_table_err= 0;
    DEBUG_ENTER("get_table_share");

    /* 先通过HASH算法，在table_def_cache，也就是table_definition_cache中找到
       已经存在的表对象TABLE_SHARE，如果这个对象正在被使用，则需要等待 */
    while ((share= reinterpret_cast<TABLE_SHARE*>(
        my_hash_search_using_hash_value(
            &table_def_cache, hash_value,
            reinterpret_cast<uchar*>(const_cast<char*>(key)),
```

```

        key_length))))
    {
        if (!share->m_open_in_progress)
            goto found;
        mysql_cond_wait(&COND_open, &LOCK_open);
    }

    /* 如果代码执行到这里, 则说明上面没有找到, 说明这个表还没有被用过,
       需要从数据字典中找到这个表, 并且转换为TABLE_SHARE */
    if (!(share= alloc_table_share(table_list, key, key_length)))
        DBUG_RETURN(0);

    /* 加载TABLE_SHARE之后, 将其加入到table_def_cache的缓存中, 通过HASH算法插入 */
    if (my_hash_insert(&table_def_cache, (uchar*) share))
    {
        free_table_share(share);
        DBUG_RETURN(0);        // return error
    }

    /* 加上引用标记 */
    share->ref_count++;        // Mark in use
    share->m_open_in_progress= true;    // Mark being opened

    /* 从数据字典中找到这个表, 将所有信息填充到SHARE对象中 */
    open_table_err= open_table_def(thd, share, db_flags);

    mysql_mutex_lock(&LOCK_open);
    share->m_open_in_progress= false;
    mysql_cond_broadcast(&COND_open);

    /* 加载并处理完成, 返回SHARE对象 */
    DBUG_RETURN(share);

found:
    ++share->ref_count;
    DBUG_RETURN(share);
}

```

但是如果这个表之前已经被访问过了, 那么就不需要再像上面一样构造这个共享结构体了, 而是直接通过 HASH 的 Key 值在 table_def_cache 中找到这个共享结构体即可。

从上面的叙述中知道，当系统得到一个 SHARE 对象之后，系统会重新再构造一个新的对象交给当前的操作，而这个对象肯定不是 TABLE_SHARE，因为它是缓存对象，是静态的、只读的，真正与用户交互的是 TABLE_SHARE 的一个衍生品，它对应的结构体名字为 TABLE，它是真正在操作中被使用的对象。那是如何从 TABLE_SHARE 变为 TABLE 的呢？可以先把这个构造过程称为实例化。先来看精简的代码片段，如下。

```
int open_table_from_share(THD *thd, TABLE_SHARE *share, const char *alias,
                        uint db_stat, uint prgflag, uint ha_open_flags,
                        TABLE *outparam, bool is_create_table)
{
    memset(outparam, 0, sizeof(*outparam));
    outparam->in_use= thd;
    outparam->s= share;
    outparam->db_stat= db_stat;
    outparam->write_row_record= NULL;

    /* Allocate handler */
    outparam->file= 0;
    /* 通过调用下面即将介绍的存储引擎句柄创建接口create，来获得一个表实例化句柄，
       后面有函数get_new_handler的实现代码，简单看看就都明白了 */
    if (!(outparam->file= get_new_handler(share, &outparam->mem_root,
                                         share->db_type())))
        goto err;

    /* 实例化之后需要存储一些动态的内容，比如插入、更新等操作时，
       需要存储当前被插入的记录，这里就是用来申请一条记录的存储空间 */
    if (!(record= (uchar*) alloc_root(&outparam->mem_root,
                                     share->rec_buff_length * records)))
        goto err;
    outparam->record[0]= record;

    /* 从SHARE对象中，COPY每一个列的信息到outparam，outparam就是实例化
       线程独享的表结构对象 */
    if (!(field_ptr = (Field **) alloc_root(&outparam->mem_root,
                                             (uint) ((share->fields+1)*
                                                         sizeof(Field*)))))
        goto err;
    /* purecov: inspected */

    outparam->field= field_ptr;
    record= outparam->record[0]-1;    /* Fieldstart = 1 */
}
```

```

if (share->null_field_first)
    outparam->null_flags= record+1;
else
    outparam->null_flags= (record+ 1+ share->reclength -
                          share->null_bytes);

/* Setup copy of fields from share, but use the right alias and record */
for (i=0 ; i < share->fields; i++, field_ptr++)
{
    Field *new_field= share->field[i]->clone(&outparam->mem_root);
    *field_ptr= new_field;
    if (new_field == NULL)
        goto err;
    new_field->init(outparam);
    new_field->move_field_offset((my_ptrdiff_t) (outparam->record[0] -
                                                outparam->s->default_values));
}

/* 处理索引的COPY, 将这些信息也实例化 */
/* Fix key->name and key_part->field */
if (share->key_parts)
{
    KEY *key_info, *key_info_end;
    KEY_PART_INFO *key_part;
    uint n_length;
    n_length= share->keys * sizeof(KEY) +
              share->key_parts * sizeof(KEY_PART_INFO);

    if (!(key_info= (KEY*) alloc_root(&outparam->mem_root, n_length)))
        goto err;
    outparam->key_info= key_info;
    key_part= (reinterpret_cast<KEY_PART_INFO*>)(key_info+share->keys);
    memcpy(key_info, share->key_info, sizeof(*key_info)*share->keys);
    memcpy(key_part, share->key_info[0].key_part, (sizeof(*key_part) *
                                                    share->key_parts));

    for (key_info_end= key_info + share->keys ;
         key_info < key_info_end ;
         key_info++)
    {
        KEY_PART_INFO *key_part_end;

```



```

key_info->table= outparam;
key_info->key_part= key_part;

for (key_part_end= key_part + key_info->actual_key_parts ;
     key_part < key_part_end ;
     key_part++)
{
    Field *field= key_part->field= outparam->field[key_part->fieldnr-1];

    if (field->key_length() != key_part->length &&
        !(field->flags & BLOB_FLAG))
    {
        /*
         * We are using only a prefix of the column as a key:
         * Create a new field for the key part that matches the index
         */
        field= key_part->field=field->new_field(&outparam->mem_root,
                                                outparam, 0);

        field->field_length= key_part->length;
    }
}
/* Skip unused key parts if they exist */
key_part+= key_info->unused_key_parts;
}
}

/* Fill record with default values */
if (outparam->record[0] != outparam->s->default_values)
    restore_record(outparam, s->default_values);

/* Increment the opened_tables counter, only when open flags set. */
/* 这个操作，就是要更新状态参数Opened_tables */
if (db_stat)
{
    /* 上面已经对这个表做了初始化实例化，此时就将这个表打开，打开表所做的
       的操作包括初始化表的自增列值，在内存中创建表的索引对象以及创建表
       在数据搜索时的元组结构等 */
    if ((ha_err= (outparam->file->
                    ha_open(outparam, share->normalized_path.str,
                            (db_stat & HA_READ_ONLY ? O_RDONLY : O_RDWR),
                            (db_stat & HA_OPEN_TEMPORARY ? HA_OPEN_TMP_TABLE :

```

```

        ((db_stat & HA_WAIT_IF_LOCKED) ||
         (specialflag & SPECIAL_WAIT_IF_LOCKED)) ?
        HA_OPEN_WAIT_IF_LOCKED :
        (db_stat & (HA_ABORT_IF_LOCKED | HA_GET_INFO)) ?
        HA_OPEN_ABORT_IF_LOCKED :
        HA_OPEN_IGNORE_IF_LOCKED) | ha_open_flags))))
    {
        /* error handle */
        goto err;
    }
}
thd->status_var.opened_tables++;

DEBUG_RETURN (0);
}

```


从代码中可以看到, 其实这两个结构体的很多成员是相同的, 并且可以直接复制过去, 因为 TABLE_SHARE 是一个静态的缓存对象, 所以相对而言, TABLE 就可以称作一个相对动态的、正在进行一些操作的实例了。TABLE 中有一个成员就是直接指向了 TABLE_SHARE; 还有一些成员比如 record, 是用来构造插入操作中的一条记录的, 在实例化的时候, 会根据这个表定义的每一个列及其数据类型等提前构造好; field 用来存储这个表中所有的列信息, 这些信息其实是完全将 SHARE 中的信息克隆过来的。其他的一些小细节就不叙述了, 不过还有两个很重要的东西必须要说一下。

第一, 因为上面已经提到了, TABLE 这个对象是一个动态的、被实例化的对象, 它相当于是一个被打开的表, 它已经不仅仅是 MySQL Server 层的对象了, 而是具体到某一个存储引擎了, 所以这里还需要构造这个对象有关存储引擎的信息, 并且打开这个表。

第二, 因为 MySQL 是一个插件式的数据库管理系统, 对于表对象的管理, MySQL 层与存储引擎层就是在这里分开的。TABLE 算是它们之间的桥梁, 下层是存储引擎, 上层就是 MySQL 了。对于 MySQL 的存储引擎, 都要提供一些公共的接口来驱动其存储引擎, 这些接口都是给上层调用, 来操作对应的存储引擎的, 也可以被称作 MySQL 与存储引擎之间交流的通道。

从上面的函数 open_table_from_share 中看到, 实例化的时候, 通过调用 get_new_handler 函数来得到要打开表的句柄, 它在内部调用了存储引擎接口 create, 在内存中创建了一个实例化的表对象, 精简代码如下。

```

 handler *get_new_handler(TABLE_SHARE *share, MEM_ROOT *alloc,
                           handlerton *db_type)
{
    handler *file;

```

```

if (db_type && db_type->state == SHOW_OPTION_YES && db_type->create)
{
    /* 创建、初始化接口innobase_create_handler */
    if ((file= db_type->create(db_type, share, alloc)))
        file->init();
    DEBUG_RETURN(file);
}
}

```

实例化过程，首先需要调用函数 `create` 来创建一个对应的存储引擎句柄，创建之后就通过这个表句柄的接口函数 `ha_open`（在 `open_table_from_share` 的最下面可以看到这个函数的调用）来打开这个表。这里需要注意的是，这两个接口是属于不同层次的，`create`（这里对应的是 `innobase_create_handler`）是存储引擎层面的接口，可以通过它在内存中创建一个缓存对象，这个对象与参数 `share` 关联起来，之后在打开这个表的时候，使用这个表的接口函数 `ha_open` 打开，也就是说 `ha_open` 是针对一个表而言的，通过 `create` 创建句柄之后，再通过 `ha_open` 打开。

打开表主要是对这个新创建的存储引擎句柄进行一些初始化操作。在打开之后，这个表的实例化也就算完成了，而这个已经被打开的实例句柄就挂在 `TABLE` 结构体中。从这里也可以看出，`TABLE` 对象与一个 MySQL 操作相对应，在 MDL 锁不冲突的情况下，每一个线程对应一个实例化的表对象 `TABLE`，而它们指向的是同一个 `SHARE` 对象。

在实例化完成之后，还涉及一个实例化表对象缓存空间，对应的参数是 `table_open_cache`。实例化之后，会将实例化的表对象加入到这个缓存中，代码如下。

```

bool Table_cache::add_used_table(THD *thd, TABLE *table)
{
    Table_cache_element *el;

    /* 因为在5.6及以上的版本中，表实例对象缓存，已经是多实例的方式管理的。
       为了提高并发度，和多实例的Buffer Pool有点类似，那么这里是先找到当前
       表要被缓存的实例 (this)，也就是要被HASH到哪一个实例中 */
    el= table->s->cache_element[table_cache_manager.cache_index(this)];

    if (!el)
    {
        /* 如果没有找到，就新建一个缓存对象，缓存所依托的对象是SHARE。
           SHARE对象建立好之后，才会把实例化的TABLE对象挂在这个上面，即放
           在push_front中。从下面的代码中可以看到，只要这个表的SHARE对象存在了，
           以后就不会再新建，而是直接将另一个新实例化的TABLE继续挂在这个

```

```

        SHARE对象下面。*/
    if (!(el= new Table_cache_element(table->s)))
        return true;
    /* 插入到当前表实例缓存空间中 */
    if (my_hash_insert(&m_cache, (uchar*)el))
    {
        delete el;
        return true;
    }
    table->s->cache_element[table_cache_manager.cache_index(this)]= el;
}

/* Add table to the used tables list */
el->used_tables.push_front(table);
m_table_count++;
/* 这里会进行检查。如果表的个数, 超过参数设置大小, 便会从缓存空间中淘汰一些出去 */
free_unused_tables_if_necessary(thd);
return false;
}

```

下面来看函数 `free_unused_tables_if_necessary` 是如何淘汰表对象的。



```

void Table_cache::free_unused_tables_if_necessary(THD *thd)
{
    /* table_cache_size_per_instance参数是根据table_open_cache
       参数及table_cache_instances参数算出来的, 用来表示在每一个
       缓存实例中, 最大可以存储多少个表实例对象。因为目前是多实例的
       实现方式, 淘汰操作是每一个实例自己单独管理, 所以就
       只能参照参数table_cache_size_per_instance了。
       如果当前实例中的表个数, 已经大于最大值, 则开始淘汰, 直到
       当前实例中的表个数小于最大值为止。
    */
    if (m_table_count > table_cache_size_per_instance && m_unused_tables)
    {
        mysql_mutex_lock(&LOCK_open);
        while (m_table_count > table_cache_size_per_instance &&
               m_unused_tables)
        {
            TABLE *table_to_free= m_unused_tables;
            remove_table(table_to_free);
            intern_close_table(table_to_free);
            /* 统计状态参数table_open_cache_overflows的值, 表示被淘汰了多少个表。

```

```

        这个值可以在私下测试观察其具体表现 */
        thd->status_var.table_open_cache_overflows++;
    }
    mysql_mutex_unlock(&LOCK_open);
}
}

```

在被实例化之后，这个表对象就可以直接与存储引擎进行交互。比如插入一条记录，直接调用 TABLE 已经被实例化的存储引擎句柄的接口函数 `ha_write_row` 即可。

当一个操作完成之后，它所实例化的表就不需要了，此时系统不是将这个本地的实例直接释放掉，而是将其保存下来。保存下来是为了下次某一个用户再次访问这个表的时不需要再次进行实例化，直接拿过来用即可，当然，可能需要一些额外的操作，比如将实例状态恢复，调用函数 `ha_reset` 即可。

系统在保存一个不使用的实例化对象时，是直接将其放在 SHARE 的一个 `free_tables` 链表中，但首先要从 `used_tables` 链表上摘下来，这两个链表都是用来保存这个表的所有实例的，`used_tables` 用来存储正在使用的实例，`free_tables` 用来存储所有当前未使用的实例。在并发比较高的情况下，可能在 `used_tables` 中有多个，在 `free_tables` 中却没有，但在全部执行完成之后则相反，那么如果此时有用户再操作这个表，系统可以直接从 `free_tables` 中找一个来用即可。

现在可以知道，在 MySQL 中，表对象的缓存其实是用两部分。一部分是 SHARE 的缓存，也就是多个不同表的 SHARE 对象的缓存；另一部分就是每一个 SHARE 结构被实例化之后的实例对象的缓存，MySQL 用来管理缓存空间大小的方法是通过计数来实现的。默认情况下，系统中总的 SHARE 个数不能超过 `table_definition_cache`（对应参数 `table_definition_cache`）个。

上面提到的都是关于表对象 SHARE 结构的缓存，既然是缓存，肯定有它相应被删除或淘汰的问题，当然在这里也不例外。那么，在什么情况下 SHARE 结构会被淘汰或删除呢？很明显，如果只是对这个表进行增删改等没有涉及修改表定义的操作，SHARE 是不会被删除的，只有可能会被淘汰，因为如果查询太多表的话，表对象缓存个数是有限制的，当到达这个数目之后，系统会自动将一些不经常使用的 SHARE 淘汰掉，这很容易理解。

一般情况下，只要对表结构、依赖关系、表定义等方面进行过修改，这个表对象的缓存 SHARE 对象就必须要从缓存中删除，同时要删除它上面所有被实例化的表对象缓存结构，因为这个表的版本被更新了，如果继续将其缓存的话，是不安全的，或者是错误的，又或者会导致一些不可预知的问题。这样，其他用户等待表对象的修改操作完成之后（因为修改过程中这个表是被上了锁的，进行操作需要等待），会又一次像前面所讲的一样，首先是从缓存中找这个表的缓存对象，如果找不到的话，再从数据字典（系统表）中读取进来，然后继续操作。

涉及的参数变量

与表对象缓存相关的参数,包括两个,分别是 `table_open_cache` 和 `table_definition_cache`。这一节详细讲述它们之间的关系及需要注意的一些细节。

前面已经讲到,表的缓存实际上是有两层的。从文件开始往上,第一层是数据字典的缓存,是通过 `.frm` 文件内存化之后,被缓存起来的对象,也就是上面所说的 `SHARE` 对象的缓存,其空间大小通过参数 `table_definition_cache` 来控制,以表的个数为单位。第二层就是将这些 `SHARE` 对象实例化并打开之后表所占用的缓存空间,其大小通过参数 `table_open_cache` 来控制,以表的实例化个数为单位。

如果一个实例中的表非常多,则可以通过设置一个比较大的表定义缓存空间来加速对表的处理。现在可以看到,这个缓存空间所缓存的 `SHARE` 对象,实际上比已打开表的缓存对象 `TABLE` 占用空间小一些,这是因为一个 `SHARE` 对象可以被多个线程实例化。

`table_definition_cache` 参数的最小值为 400,不过这个参数的设置与 `table_open_cache` 之间有一定的关系,默认情况下,它们之间的换算方法如下。

```
void adjust_table_def_size()
{
    ulong default_value;
    sys_var *var;

    /* 最大值为2000,最小值为400。
       table_cache_size对象的值就是参数table_open_cache的值 */

    default_value= min<ulong> (400 + table_cache_size / 2, 2000);
    var= intern_find_sys_var(StringWithLen("table_definition_cache"));
    DBUG_ASSERT(var != NULL);
    var->update_default(default_value);

    /* 得到参数table_definition_cache的值为default_value (table_def_size) */
    if (! table_definition_cache_specified)
        table_def_size= default_value;
}
```

从代码可以看到, `table_definition_cache` 通过这样的方法计算时,最大值为 2000,也就是说,如果 `table_open_cache` (对应代码中的 `table_cache_size`) 大于 3200 (因为 $3200/2+400$ 的值为 2000), 则 `table_definition_cache` 就不会再跟着 `table_open_cache` 的增长而增长了。

而参数 `table_cache_size`, 也是通过一系列复杂的计算得来的, 它与参数 `innodb_open_files`、`max_connections` 等都有关系, 下面再看一下这些参数的计算方式。

先来看这些参数的计算逻辑顺序关系，如下。

```
void adjust_related_options(ulong *requested_open_files)
{
    /* In bootstrap, disable grant tables (we are about to create them) */
    if (opt_bootstrap)
        opt_noacl= 1;

    /* The order is critical here, because of dependencies. */
    /* 先计算open_files_limit, 后面在此基础上再计算max_connections */
    adjust_open_files_limit(requested_open_files);
    adjust_max_connections(*requested_open_files);

    /* 在上面已知max_connections、open_files_limit的基础上,
       计算参数table_open_cache*/
    adjust_table_cache_size(*requested_open_files);

    /* 上面table_definition_cache参数的计算方法 */
    adjust_table_def_size();
}
```

adjust_related_options 的调用，其实是在启动函数 mysql_main 中调用的，该启动函数在第2章有讲过。

计算 open_files_limit 的方法如下。

```
void adjust_open_files_limit(ulong *requested_open_files)
{
    ulong limit_1;
    ulong limit_2;
    ulong limit_3;
    ulong request_open_files;
    ulong effective_open_files;

    /* 通过下面三种方式的计算，选择合适的值出来 */
    /* MyISAM requires two file handles per table. */
    /* 参数extra_max_connections为MySQL系统参数extra_max_connections的值 */
    /* 参数table_cache_size为调整前的参数table_cache_size的值,
       之后还会在下面的函数中调整 */
    limit_1= 10 + max_connections + extra_max_connections + table_cache_size * 2;

    /*
```



```

    We are trying to allocate no less than max_connections*5 file
    handles (i.e. we are trying to set the limit so that they will
    be available). 每一个连接, 预计平均会打开5个文件
    */
    limit_2= (max_connections + extra_max_connections) * 5;

    /* Try to allocate no less than 5000 by default. */
    limit_3= open_files_limit ? open_files_limit : 5000;

    /* 选择三种方法预计出来的最大值为理论上的open_file_limit, 因为
       还需要等待设置成功, 这个值才算生效, 不然只是如变量名所表现的
       request的值, 而不是实际的open_file_limit */
    request_open_files= max<ulong>(max<ulong>(limit_1, limit_2), limit_3);

    /* Notice: my_set_max_open_files() may return more than requested. */
    effective_open_files= my_set_max_open_files(request_open_files);

    open_files_limit= effective_open_files;
    if (requested_open_files)
        *requested_open_files= min<ulong>(effective_open_files, request_open_files);
}

```

计算 max_connections 的方法, 如下。

```

void adjust_max_connections(ulong requested_open_files)
{
    ulong limit;

    /* TABLE_OPEN_CACHE_MIN的值为400 */
    limit= requested_open_files - 10 - TABLE_OPEN_CACHE_MIN * 2;

    /* 根据之前计算出来的requested_open_files值, 再反过来调整max_connections的值 */
    if (limit < max_connections)
        max_connections= limit;
}

```

计算 table_open_cache 的方法, 如下。

```

adjust_table_cache_size(ulong requested_open_files)
{
    ulong limit;

```

```

limit= max<ulong>((requested_open_files - 10 - max_connections) / 2,
    TABLE_OPEN_CACHE_MIN);

/* 根据之前计算出来的requested_open_files值, 再反过来调整table_open_cache的值 */
if (limit < table_cache_size)
    table_cache_size= limit;
}

```

优缺点总结

本节所讲述的 MySQL 表的缓存机制有如下很多优点。

- 相比全字典缓存（全字典缓存的意思是在数据库启动时将所有的数据字典信息都一次性载入到内存中来，这样在使用过程中的效率非常高，但在 DDL 操作方面有很大的不足），它的触发时机是用到的时候才载入缓存，在被修改之后，会将其从缓存中删除，以后用到的时候，再次载入，这样的实现方式降低了 DDL 操作或回滚导致的字典缓存维护工作的代价。
- 有效地利用了内存空间，因为可以通过设置表对象缓存空间的大小来控制内存的使用情况，同时只有用到的对象才会被载入到内存中，提高了内存的利用率。

上面所述的 MySQL 表缓存实现方案虽说还是比较先进的，但也有一些缺点，如下。

- 该缓存实现方案在效率方面还是有些优化空间的。比如上面提到的，控制缓存空间大小是根据实例化表对象的个数来计算的，在系统中默认最大值是 `table_open_cache`，如果超过这个值系统会自动淘汰一些不常用的实例化表对象。但是如果一个表的定义非常大，那么在并发情况下，就有可能建立很多个实例化表对象，假设对象个数接近 `table_open_cache`，那么这样算下来有可能会将操作系统的内存用光，这是不可控制的，也是不可预期的。对于 `SHARE` 的缓存也是一样，如果一个用户访问了很多不同的定义或很大的表，也会有同样的问题。
- 从前面的讲述也可以看出，为了实现插件式的数据库，其实还是有一些效率的代价的。在表的缓存方面，中间加入了一层 `SHARE` 的缓存，真正用到的时候还需要实例化，因为每一个用户的操作及不同时间的状态都是不同的，所以每一个用户必须要在 `SHARE` 的基础上实例化一个新的对象出来，这样就给内存、CPU 带来了一定程度上的浪费及压力。

存在的问题

此外，该缓存方案也存在一些问题，如下。

- SHARE 缓存：个人认为有一个更好的办法来很精确地通过具体的空间大小来管理表缓存空间。因为 SHARE 缓存对象是静态的，是个结构体，通过使用计数来控制内存的使用，有可能会造成内存用光的情况。那么对于 SHARE 对象，完全可以把它流式化（扁平化），也就是说把这个结构体的大小计算出来以后，申请相应的空间，将结构体中的所有信息都按照固定的顺序写入这块内存中，这样，一个 SHARE 所占的空间大小就固定了，便可以完全通过设置表缓存空间大小来管理表对象缓存了，那么上面提到的内存用光的问题就自然解决了。当然，这个缓存空间大小需要根据计算机的内存大小进行合理的设置，避免出现不可预料的问题。
- TABLE 缓存：TABLE 实例的缓存同样存在上面的问题，解决方案与上面的思想差不多。因为这个对象是一直被用的，它是一个实例，所以就不能直接像上面一样，将其流式化，而是可以通过申请一片连续的空间，这个实例中的所有指针或其成员的值都指向（有可能要对齐）这个空间中的指定位置，如此，这个结构体的使用没有任何改变，但其占用的空间大小是固定的，同样可以通过用户手动设置 TABLE 实例缓存空间的大小来管理缓存空间，也避免了表定义太大导致内存用光的问题。

5 InnoDB 初探

前面主要讲述了 MySQL 服务器层的一些东西，包括服务器启动、用户线程创建、表对象、客户端请求处理及源代码目录结构等，剩下的一个很重要的模块就是存储引擎。这里主要讲述的是 InnoDB 的一些重要内容，包括 InnoDB 的源代码目录结构、InnoDB 存储引擎的体系结构、文件系统、存储引擎启动关闭、数据字典及表对象缓存等内容。

InnoDB 的源代码目录结构

之前已经讲过，源码目录下的 `storage` 目录就是关于存储引擎的，里面有一个 `innobase` 目录，其主要目录结构如图 5.1 所示，目录名解释说明如下表。

目录名	解释说明
btr	在这个目录中，实现了所有关于 B+ 树的功能，包括创建销毁、搜索、增删改查等
buf	在这个目录中，实现了所有关于 buffer 缓存的功能，包括载入、淘汰、刷盘等
dict	实现了 InnoDB 数据字典的加载、存储、修改及内存管理等功能
fil	实现了 InnoDB 中，关于物理文件的读写、内存对象管理、文件扩展等功能
fsp	包含所有 InnoDB 中物理文件内部空间的管理，包括数据段、物理页面及簇
ibuf	实现了所有关于 insert buffer 的功能，不过在最新版本里已经不只是 insert 了，还包括了删除及更新操作，最新名字叫 change buffer

```
drwxr-xr-x 40 zhufeng staff 1360 Mar 7 14:28 ./
drwxr-xr-x 14 zhufeng staff 476 Mar 7 14:27 ../
-rw-r--r-- 1 zhufeng staff 10432 Mar 7 14:28 CMakeLists.txt
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 api/
drwxr-xr-x 6 zhufeng staff 204 Mar 7 14:28 btr/
drwxr-xr-x 10 zhufeng staff 340 Mar 7 14:28 buf/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 data/
drwxr-xr-x 9 zhufeng staff 306 Mar 7 14:28 dict/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 dyn/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 eval/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 fil/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 fsp/
drwxr-xr-x 16 zhufeng staff 544 Mar 7 14:28 fts/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 fut/
drwxr-xr-x 5 zhufeng staff 170 Mar 7 14:28 ha/
-rw-r--r-- 1 zhufeng staff 104 Mar 7 14:28 ha_innodb.def
drwxr-xr-x 7 zhufeng staff 238 Mar 7 14:28 handler/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 ibuf/
drwxr-xr-x 229 zhufeng staff 7786 Mar 7 14:28 include/
drwxr-xr-x 5 zhufeng staff 170 Mar 7 14:28 lock/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 log/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 mach/
drwxr-xr-x 5 zhufeng staff 170 Mar 7 14:28 mem/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 mtr/
drwxr-xr-x 6 zhufeng staff 204 Mar 7 14:28 os/
drwxr-xr-x 5 zhufeng staff 170 Mar 7 14:28 page/
drwxr-xr-x 11 zhufeng staff 374 Mar 7 14:28 pars/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 que/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 read/
drwxr-xr-x 4 zhufeng staff 136 Mar 7 14:28 rem/
drwxr-xr-x 18 zhufeng staff 612 Mar 7 14:28 row/
drwxr-xr-x 6 zhufeng staff 204 Mar 7 14:28 srv/
drwxr-xr-x 5 zhufeng staff 170 Mar 7 14:28 sync/
drwxr-xr-x 10 zhufeng staff 340 Mar 7 14:28 trx/
drwxr-xr-x 3 zhufeng staff 102 Mar 7 14:28 usr/
drwxr-xr-x 13 zhufeng staff 442 Mar 7 14:28 ut/
```

图 5.1

续表

目录名	解释说明
lock	实现了在 InnoDB 中所有关于行锁及表锁的功能
log	实现了所有关于 InnoDB 日志相关的功能, 包括写日志及数据恢复
mem	实现了在 InnoDB 内部用来内存管理的功能
mtr	一个很重要的模块, 实现了连接上层事务与下层物理文件及日志的桥梁
os	封装了不同操作系统下, 与操作系统交互的功能, 包括同步、文件及线程等
page	实现了关于数据页面内数据管理的功能, 包括插入、删除等
pars	包含了 InnoDB 内部自己实现的语法、语义分析器
que	包含了 InnoDB 内部自己实现的一个状态机执行器
rem	包含了在 InnoDB 中对物理记录的管理, 包括记录比较及转换计算等
row	包含了各种物理记录的操作实现, 包括增删改查、回滚、purge、合并等
srv	这里实现了 InnoDB 系统后台的管理, 著名的 master 线程就在这里面实现
sync	实现了 InnoDB 读写同步的功能, 包括等待队列、读写锁及 mutex 等实现
trx	实现了有关事务的功能, 包括 MVCC、回滚段、purge、回滚记录及回滚提交等
ut	这里实现了很多内部使用的一些经典算法, 包括链表、红黑树、堆排序等

InnoDB 存储引擎文件组织

使用过 MySQL 及 InnoDB 存储引擎的同学应该比较清楚 InnoDB 的文件组织结构。下面首先看一下它的文件（包括 MySQL 文件）结构示意图，如图 5.2 所示。

图 5.2 中所示都是假设将所有 MySQL 文件配置放在一个目录下。

最左边的是一些 MySQL 的日志文件，包括如下 3 个文件。

- slow.log 文件，会记录慢查询日志，当一条语句执行时间超过在配置参数 `long_query_time` 中指定的值时，这条语句就会被记录在这个文件中。
- error.log 文件，会记录一些系统启动或运行时的错误或警告信息，通过配置参数 `log_error` 来设置。
- general.log 文件，会记录所有在数据库上执行的语句，经常用来追踪问题，但会影响一点性能，所以一般不会打开，只有在调试的时候会偶尔开启。当然，在系统 QPS 不大的情况下，还是可以打开的，因为查找问题时，问题已经发生了，不好复现，所以 `general_log`

是一个很好的查找问题的途径。只是在 QPS 很高的情况下，这个文件有可能会非常大，不太好处理，所以需要自己权衡利弊。

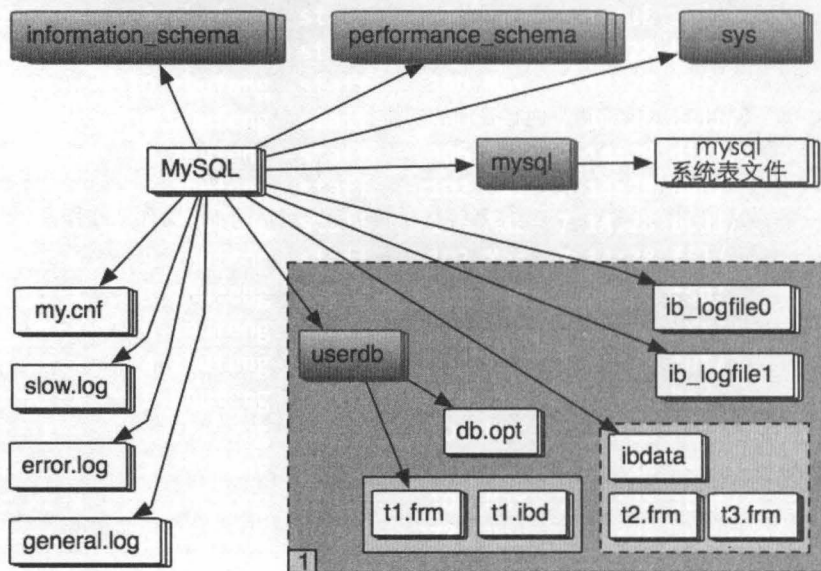


图 5.2

在 MySQL 系统的 datadir 目录下，还有一个目录叫 mysql。在 MySQL 数据库系统中，一个数据库就是一个目录，所以如果用户手动在 datadir 目录下创建一个目录的话，在执行 show databases 语句时，也会被显示出来。这个 mysql 目录实际上是 MySQL 数据库的一些系统表，比如权限、用户等，但这些表都是 MyISAM 存储引擎的表。不过据说在 MySQL 8.0 版本中，系统表做了翻天覆地的大改变，我们所熟悉的 mysql 数据库会不复存在，同样，熟知的.frm 文件也将被去除，这些信息都会被存储到 ibdata 中去。虽然，系统表改变之后，一些熟知的信息不存在了，但我自己看来，倒是一点都不会想念，因为这样的改变会给运维带来极大的好处，算是 MySQL DBA 的福音，但也别太乐观，我们还会处于并将长期处于一个过渡阶段。

从图 5.2 的最上方可以看到，有三个数据库，分别如下。

information_schema

information_schema 数据库是 MySQL 自带的，它提供了访问数据库元数据的方式，也就是通常所说的 Metadata。Metadata 是关于数据的数据，如数据库名或表名、列的数据类型或访问权限等，用于表述该信息的其他术语包括“数据词典”和“系统目录”。

在 MySQL 中, `information_schema` 被看作是一个数据库, 确切地说是信息数据库。其中保存着关于 MySQL 服务器维护的所有其他数据库的信息, 如数据库名、数据库表、表列的数据类型及访问权限等。在 `information_schema` 中, 有数个只读表。它们实际上是视图, 而不是基本表, 因此, 无法看到与之相关的任何文件。如果想看它包含什么信息的话, 只需要进入这个数据库, 然后一个个地执行 `show create table tablename` 即可, 这样就可以看到每一个表的功能了。

这个库在很多情况下, 可以帮助我们做一些自动化处理的工作, 比如巡检程序找到所有的 MyISAM 表, 或者是找到所有的自增列快达到上限的表, 等等。更重要的是, 这是很多工具的信息来源, 比如我们所熟知的 Percona Toolkit 工具包, 它在查一些信息的时候也是通过在这个库上执行 SQL 语句来做的。

但是, 这个库也有其明显的缺点。它在每次查找的时候, 都会现场统计相应的信息, 这需要将相应信息加载到内存中, 做成内存表, 然后将信息返回给客户端, 但如果表比较多的话, 这些语句的执行会非常慢, 造成一些不可预知故障的风险, 所以使用这个库的话, 需要谨慎处理。

关于内存表的问题, 可以通过 `show create table tablename;` 语句看到, 表的存储引擎为 MEMORY。举一个例子, 如下。



```
CREATE TEMPORARY TABLE `INNODB_LOCK_WAITS` (
  `requesting_trx_id` varchar(18) NOT NULL DEFAULT '',
  `requested_lock_id` varchar(81) NOT NULL DEFAULT '',
  `blocking_trx_id` varchar(18) NOT NULL DEFAULT '',
  `blocking_lock_id` varchar(81) NOT NULL DEFAULT ''
) ENGINE=MEMORY DEFAULT CHARSET=utf8
```

针对监控方面, 有很多表都是对 `show status` 结果的一个汇总, 使得监控变得相对更容易一些, 比如表 `INNODB_METRICS` 就是其中一个。

performance_schema

`performance_schema` 数据库是在 MySQL 5.5 中新增加的, 命名为 `performance_schema`, 从名字中也可以初步了解到, 它是针对性能的, 主要用于收集数据库服务器性能参数。`performance_schema` 提供如下功能。

- 提供进程等待的详细信息, 包括锁、互斥变量、文件信息。
- 保存历史事件汇总信息, 为判断 MySQL 服务器性能做出详细的依据。
- 添加或删除监控事件点都非常容易, 并可以随意改变 MySQL 服务器的监控周期。

sys

sys 数据库是 MySQL 5.7 中首次加入的一个系统信息库, 这个库类似 Oracle 中的动态视图, 通过这个库可以快速地了解系统的元数据信息, 并非常方便地让 DBA 发现数据库的很多信息, 在解决性能瓶颈、自动化运维等方面都提供了巨大的帮助。这个库在 MySQL 5.7 中是默认存在的, 在 MySQL 5.6 版本及以上可以手动导入, 数据库包可以在 Github 上自行下载并导入, 但功能可能并不太完善, 还是需要系统内部做一些支持。

这个库是通过视图的形式把 information_schema 和 performance_schema 结合起来的, 查询出让人更加容易理解的结果。但前提是, sys 库本身的信息来源要依赖 information_schema, 而 information_schema 又有自身的缺点, 所以使用 sys 的时候也需要格外注意一下, 在了解的情况下再使用, 以确保万无一失。

现在, 再回到图 5.2 中。图中 1 框中所显示的部分, 就是 InnoDB 存储引擎自己的东西了, 包括默认的两个日志文件, 日志文件大小通过参数 innodb_log_file_size 来设置, 个数通过参数 innodb_log_files_in_group 来设置。这几个日志文件的大小被设置后在运行过程中都是固定不变的, 如果想要修改大小或者个数, 需要关闭数据库, 修改参数之后将原来的日志文件删除, 然后重启, 即可完成修改。

可以看到, 图 5.2 中有一个目录 userdb, 这是用户创建的数据库, 里面有三个表, 分别是 t1、t2、t3, 对应三个 .frm 文件, 但可以看到只有一个 ibd 文件, 这是因为 InnoDB 有一个参数 innodb_file_per_table 可以对创建表时是否使用单独的表空间进行设置, 很明显, t1 是在这个参数为 ON 的情况下创建的, 另外两个表使用了共享表空间, 也就是说这两个表的数据会被存储到 ibdata 中。

另外, 还有一个小文件, 就是 db.opt。这个文件存储的是 MySQL 数据库的一些配置信息, 例如编码、排序的信息, 如果在创建数据库时指定一些非默认的参数, 就会在这个文件中存储这些信息。

InnoDB 体系结构

InnoDB 是 MySQL 众多存储引擎中比较流行的一个, 现在已经成为 MySQL 默认的存储引擎。目前, 在全球知名的大公司中都已经在大规模地使用, 并且因为它是开源的, 这些公司都有自己的增强版本。

InnoDB 是 MySQL 中支持事务安全的存储引擎, 它的设计实现方式与 Oracle 比较相似, InnoDB 的体系结构比较清晰, InnoDB 体系结构示意图如图 5.3 所示。

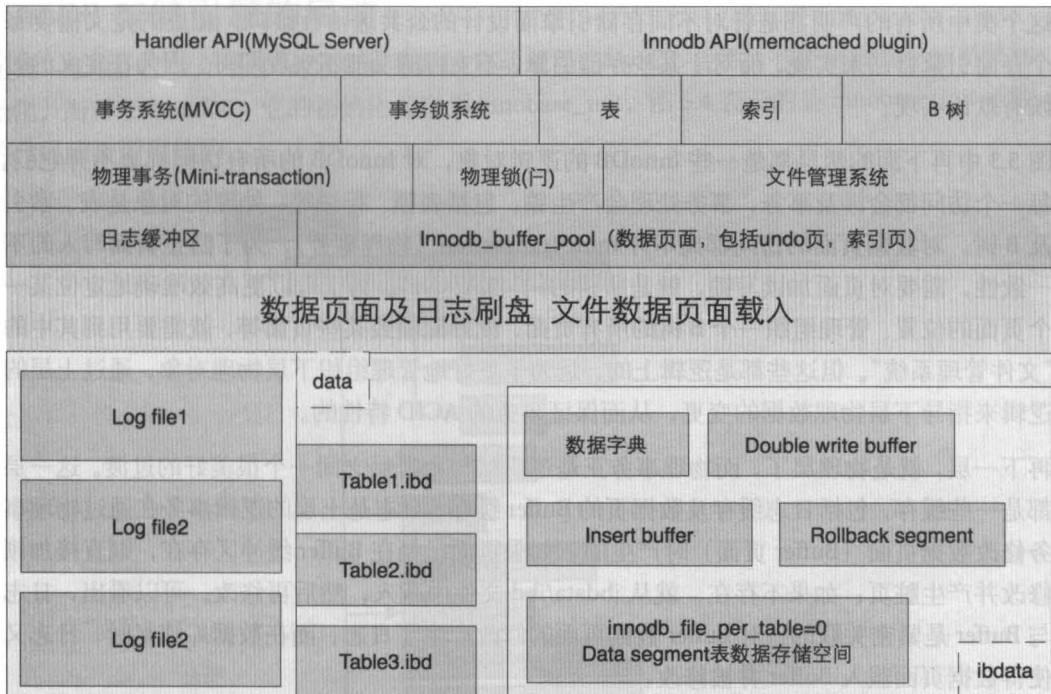


图 5.3

图 5.3 中最上层部分是提供给 MySQL Server 及 InnoDB NoSQL 的接口, 因为存储引擎相对 Server 层在下层, 通过 Server 与 Storage Engine 之间的公共接口连接起来。公共接口对应的接口定义在文件 sql/handler.h 中, 通过如下定义的类实现。

```
class handler :public Sql_alloc{
.....
virtual int rename_table(const char *from, const char *to);
virtual int delete_table(const char *name);
virtual int open(const char *name, int mode, uint test_if_locked)=0;
virtual int close(void)=0;
virtual int index_init(uint idx, bool sorted) { active_index= idx; return 0; }
virtual int index_end() { active_index= MAX_KEY; return 0; }
virtual int write_row(uchar *buf __attribute__((unused)));
virtual int update_row(const uchar *old_data __attribute__((unused)),
                      uchar *new_data __attribute__((unused)));
virtual int delete_row(const uchar *buf __attribute__((unused)));
.....
}
```

这个类中所有的声明都是针对不同存储引擎而设计的公共虚函数接口，相应的定义需要每个存储引擎自己来实现，而对于某些存储引擎，不支持的功能不实现即可，因为在定义的时候有默认实现。

图 5.3 中再下面的两层都是一些 InnoDB 的逻辑对象，对 InnoDB 的所有访问都离不开它们。每一个访问都会涉及事务，事务处理会产生锁，包括表锁、行锁等，处理的对象是表、索引及 B 树。对数据页面的访问都离不开 mini-transaction（物理事务），为了防止页面写入的不一致性，需要对页面加读写锁，就是所谓的 LATCH。为了可以更高效准确地定位某一个页面的位置、管理组织一个 B 树的所有页面，或分配销毁某些页面等，就需要用到其中的“文件管理系统”。但这些都是逻辑上的，是为了更好地管理组织下层物理对象，通过上层的逻辑来指导下层物理数据的变更，从而保证事务的 ACID 特性的。

再下一层，就是物理层了，而物理事务正是逻辑层与物理层之间一个很美好的过渡。这一层都是一些缓存，包括日志缓存及数据页的 Buffer 缓存。日志是上层的逻辑事务在通过物理事务修改数据页面（Buffer 页面）时产生的，如果页面已经在 Buffer 缓冲区存在，则直接加锁修改并产生脏页；如果不存在，就从 ibdata/ibd 文件中载入，然后再修改。可以看出，日志与 Buffer 是紧密关联的，对 Buffer 数据页面的修改产生了日志，而在数据库恢复时，日志又使得数据页面载入 Buffer 并被修改。

再下层可以算是操作系统的 IO 层，这一层主要就是用于处理下层物理文件与上层缓存（日志缓存及 Buffer 数据页面缓存）之间的交互，基本包括如下两方面的 IO。

- REDO 日志 IO：当日志缓冲区（一般比较小，也就几 MB，可以设置）满了，或者做了检查点（checkpoint），或者进行了逻辑事务提交（具体行为与 innodb_flush_log_at_trx_commit 参数有关系）等之后，日志必须落地刷到磁盘，这是日志的 IO。
- 数据页面的 IO：对于 Buffer 缓冲区的 IO，具体来说也包括两部分，分别是索引数据页面的 IO 和回滚段页面的 IO。当上层要使用某一个页面时，就从文件中载入这个页面，而当某一个页面不再被使用，或者系统在做检查点时，对应的页面就会被刷到磁盘文件中。

这两部分的详细讲述，请参考第 11 章。

上面介绍的 InnoDB 的所有核心模块，在本书的后续章节中都会做很详细的解释，这里只是简单提一下，把握一下整体的框架结构即可。

InnoDB 存储引擎启动与关闭

学习的时候，总是很想知道，作为一个很强大、很流行的数据库引擎，它在启动及关闭的时候究竟做了些什么？那么，现在就通过对其源码的跟踪，一步步地了解它是如何启动及关闭的。

InnoDB 存储引擎的启动

在第2章中，已经介绍过 MySQL 服务器的启动过程，MySQL 会将所有使用到的存储引擎初始化，而对于 InnoDB，它的初始化入口为 `innobase_init`，图 5.4 表示的是 `innobase_init` 函数的代码函数调用结构。

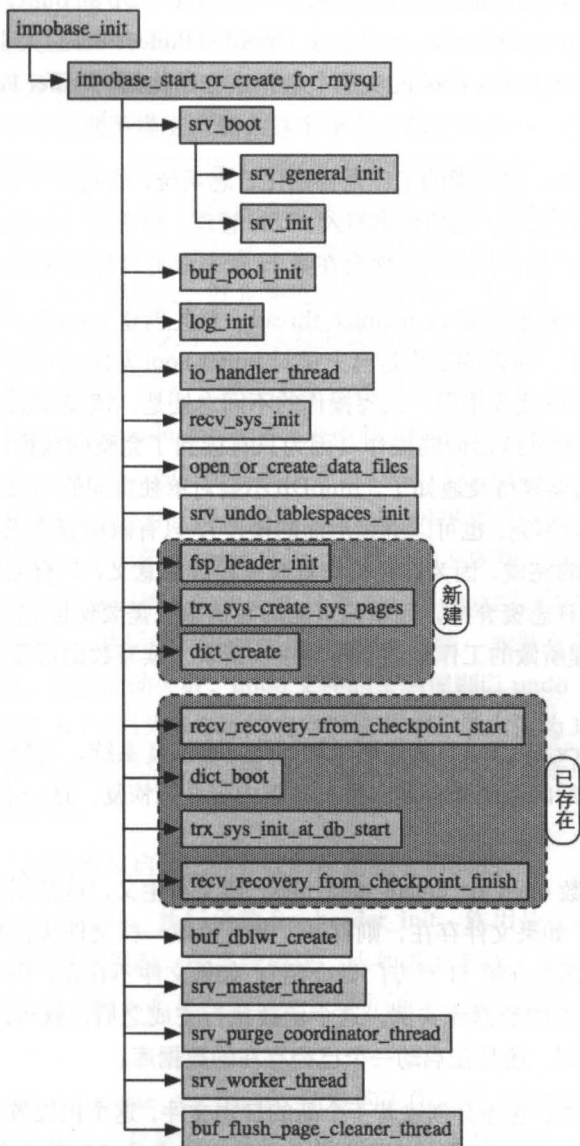


图 5.4

`innobase_init` 函数最初的作用是初始化一些全局变量，为启动做准备。

接着下面第一个很重要的函数是 `innobase_start_or_create_for_mysql`，主要完成的是 InnoDB 的启动过程，首先初始化一些系统模块，比如在函数 `srv_general_init` 中初始化了同步控制系统、内存管理系统、日志恢复变量等，`srv_init` 函数中初始化了后台线程 (`srv_sys->sys_threads`) 同步控制系统。

接下来，`buf_pool_init` 函数所做的操作就是通常所说的 InnoDB 的 Buffer Pool 的初始化，它是根据系统配置的参数 `innodb_buffer_pool_size` (InnoDB Buffer Pool 总大小) 及 `innodb_buffer_pool_instances` (InnoDB Buffer Pool 的实例个数) 来初始化的。Buffer Pool 的详细实现原理，请参考第 11 章。

再后面，是函数 `log_init`，它所做的工作是初始化日志系统，这是关于整个 InnoDB 存储引擎所有日志相关的初始化工作，包括日志写入、LSN 管理、检查点 (checkpoint point)、日志刷盘及数据恢复等操作。关于日志，后面会在第 11 章来做更详尽的讲述。

再接下来，是创建 IO 异步线程 `io_handler_thread`，总共创建 `innodb_write_io_threads + innodb_read_io_threads` 个，所做的工作是当上层对 Buffer Pool 发出读写请求时，主操作线程会将这个操作交给异步 IO 线程来做。读写操作的不同之处是，读操作需要在请求之后等待异步读的完成，然后才能继续后面的操作 (因为只有读到了完整的数据，才能继续后面的工作)；而写操作就不需要等待及通知了，InnoDB 不会对单独页面的写操作做通知及等待，因为这是没有必要的 (写不完，也可以继续后面的操作)，只有做检查点或批量刷盘操作时，才会等待这个批量操作的完成，因为这种操作具有里程碑的意义，只有完成了，确实写入文件中了，才能将对应的日志废弃掉，而单独页面的写请求只需要保证它完成即可。可以看出，这个异步 IO 处理线程所做的工作就是不断地接收请求、读写数据页面，以及在无请求时进行等待。

再接下来，是函数 `recv_sys_init`，其作用是初始化日志恢复系统。当数据库异常关闭，再次启动时，会用初始化后的系统来存储、解析日志内容并做恢复，这一点在第 11 章中会有更详尽的讲解。

再下来，就是执行函数 `open_or_create_data_files`。望文可生义，这是用来打开或创建系统数据文件 (`ibdata`) 的。如果文件存在，则打开，并且读取一些文件头信息，比如 LSN (与日志文件中的不同，后面会在第 11 章专门做介绍)；如果文件不存在，则会创建新的文件，此时相当于初始化一个新的数据库实例。这个函数执行完成之后，就可以知道当前是在初始化一个新的数据库实例，还是在启动一个已经存在的数据库。

如果是新创建的数据库，它还会创建若干个新的日志文件，这个由设置参数决定，并且将日志文件内容都初始化，然后将所有日志文件加入到日志文件系统组中管理。因为后面日志文件数量及大小都不会改变，所以加入到文件管理组后使用起来非常方便，并且在计算剩余空间大小、LSN 及刷日志时会经常用到。

如果是打开一个已经存在的数据库，则只需要将数据文件（ibdata）打开，然后打开对应的日志文件（如果存在则打开，不存在则新建），同样加入到文件管理系统中，至此，就完成了数据文件的打开或创建及文件系统的初始化了。

打开文件之后，接着是 MySQL 5.6 中才引入的一个新功能，即 `srv_undo_tablespaces_init`，其主要涉及的模块就是回滚段的存储。在 5.6 版本之前，回滚段是被强制分配在 ibdata 中的，不可以选择，也就是与 InnoDB 系统表放在一个文件中，而 5.6 版本之后，对这点做了改进，不至于因为一些大事务导致 ibdata 扩展到很大。因为回滚段存储在 ibdata 中，在 RR 隔离级别下，如果一个读事务长期不提交，那么这个事务之后的所有写事务的回滚段都不能释放空间出来，必须要存储起来，为了防止这样的事务再次读取老数据，就必须要将回滚段的数据存起来。而实际上，有时候这并非真正的需求，而是一些误操作导致的，但作为一个支持 MVCC 的关系型数据库而言，为了支持读去老数据的功能，不得不这么做。那么，这样的数据一旦因为一些大事务而不能释放，就会导致回滚段空间一直膨胀，将 ibdata 撑大，而这个文件一旦被撑大了，就很难再回去了。所以要尽量避免这样的问题出现。

根据上面有可能出现的问题，引入了将回滚段分离出来使用单独文件存储的功能。回滚段的个数通过新增参数 `innodb_undo_tableSPACE` 来控制，如果设置为 0，则使用 5.6 之前的方式将所有回滚段在 ibdata 文件中分配，而如果设置为 1~126（最大值为 126）之间的值，则会创建 `innodb_undo_tableSPACE` 个文件。但这里有一点，InnoDB 规定（由于其检查方式实现的问题），只有在创建新库时，才能新建 undo 文件，不然启动不成功。如果是启动一个已经存在的库，InnoDB 会从回滚段索引槽（后面在第 11 章中会详细介绍回滚段的实现）中读取所有使用到的 undo tablespace，然后统计并找到所有 undo 文件，如果有某一个或者多个文件打开有问题或者找不到，就会启动失败。undo 文件的全名规则以 undo 开头，后面跟着以 0 补全的文件编号，最小值为 1（因为 ibdata 的表空间值为 0），最大值为 126，例如 undo120。在找到并打开所有的 undo 文件之后，将它们全部加入到文件管理系统中，作为系统文件，这些对象在内存中是永驻的。

接下来的操作，对于新建库与启动已经存在的库是完全不同的。

先说新建库。首先初始化文件，执行函数 `fsp_header_init`，作用是在系统文件 ibdata 的一开始分配空间，以便可以存储管理一些系统模块，比如事务系统、Inode 页面（后面会在第 8 章做详细阐述）、回滚段系统页面及数据字典管理页面等。这些页面都存储在 ibdata 的头几个页面中。

下面，是执行函数 `trx_sys_create_sys_pages`，所做的工作就是上面提到的事务系统存储初始化，事务系统使用的页面为 5 号页面，也就是 ibdata 的第六个页面。这个页面存储的一个比较重要的东西就是事务 ID，因为事务 ID 在 MVCC 及事务的 ACID 管理中很重要，并且不能重复，所以这个值被固化在这个页面中。

接下来,是一个很重要的函数——`dict_create`。因为是新建库,所以需要创建新的数据字典,这里所做的操作与上面相似,首先分配一个 `ibdata` 文件中的第八个页面,用来存储数据字典使用到的几个 ID 值,分别是 ROWID、表 ID、索引 ID、当前最大表空间 ID 等。这些与前面事务 ID 比较类似,所以有相同的处理。然后,为每一个系统表创建一个 B 树,用来存储在系统启动之后,用户创建的数据库对象,比如表、索引等。最后,通过调用函数 `dict_boot` 把所有的系统表加载到内存中,以便在后面处理用户的 DDL 请求。这些表结构是常驻内存的,结构不会被修改,所以常驻内存不会有任何问题。

上面说的是新建库的内容,接着是打开已经存在的库所要做的操作。首先,第一个很重要的函数是 `recv_recovery_from_checkpoint_start` (简称 `recovery_start`),这个函数很重要的工作是扫描日志文件,将需要恢复的日志一块一块扫描出来,然后分析其完整性,将完整的日志按照页面号归类并且做 REDO 操作,这部分是 InnoDB 日志实现的核心部分,后面的第 11 章会专门来讲述日志相关的所有问题。

接下来是一个熟悉的函数 `dict_boot`,但这里与新建库不同的是,这里只将所有系统表加载到内存中,而不会创建数据字典及初始化字典存储页面。

接着是执行函数 `trx_sys_init_at_db_start`,用来初始化事务系统,并且将所有回滚段中需要处理的事务加载进来,包括 INSERT 回滚段及 UPDATE 回滚段,用来为后面的操作做准备。因为马上要执行的是 `recv_recovery_from_checkpoint_finish` (简称 `recovery_finish`),这个函数最主要的作用就是执行回滚操作 (UNDO),这里需要先说明一点,从函数 `recovery_start` 开始,到 `recovery_finish` 为止,执行顺序是有逻辑关系的。首先,`recovery_start` 是纯物理操作,因为它完全是 REDO 操作,将所有没有写入到数据页面的日志重做一遍,后面在执行 `trx_sys_init_at_db_start` 时,需要加载所有需要处理的事务,要找到每个事务的回滚段,这些数据需要在恢复完成后才可以读取,因为在这之前,这些回滚段的页面和数据页面一样也是通过 Buffer Pool 来读写的,并不能保证这些数据的正确性。

而 `recovery_finish` 在执行的时候,所处理的是逻辑的操作,因为回滚操作是针对每一个事务而言的,所以是逻辑的,需要在 `trx_sys_init_at_db_start` 执行之后,才能执行 `recovery_finish`。而与此同时,在执行 `recovery_finish` 时,需要保证的是回滚段数据的正确性,因为回滚段的读写也是通过 Buffer Pool 来实现的,所以必须要在 REDO 恢复完成之后,回滚段的数据才是完整的,才能做回滚操作,正因为这些原因,所以上面的函数必须是以它们既定的顺序来执行。

接下来的工作,不管是新建库还是打开已经存在的库,都需要做。首先是创建两次写缓存 (`double_write`),对应函数为 `buf_dblwr_create`,它在 InnoDB 中是一个比较具有 InnoDB 特色的功能,同时也是一个用来保证数据正确性的很重要的功能,第 10 章会专门来讲述两次写。

再接下来,就是创建几个很重要的系统线程。首先是我们所熟悉的 master 线程 (`srv_master_thread`)。master 线程现在在 5.6 版本中精简了很多,功能很简单,就是每隔一秒钟进行一次

后台循环，在空闲与繁忙的阶段分别做不太相同的事情，但就其根本而言是基本相同的，所做的事情主要包括：后台删除废弃表、检查日志空间是否足够、后台合并 Insert Buffer 缓存、日志刷盘、做检查点。

再接下来，就是创建线程 `srv_purge_coordinator_thread` 与 `srv_worker_thread`，这两个线程通过互相配合，来完成整个 InnoDB 系统的 PURGE 操作。它们类似生产者与消费者的关系，第一个线程为生产者，后面的工作线程是消费者，但这个需要通过配置参数 `purge_threads`，只有当它大于 0 时，才创建 `purge_threads` 个工作线程。也就是说，如果没有配置这个参数，所有的 PURGE 操作将由调度线程来完成，就是所谓的自给自足。而如果是创建多个工作线程，调度线程在产生任务之后，会从所有工作线程中找到空闲的一个，然后交给它来处理这个任务。关于具体更详尽的 PURGE 的原理及实现，将在第 11 章中进行阐述。

再接下来，是创建线程 `buf_flush_page_cleaner_thread`，这个线程的主要工作是在后台每隔一秒钟，试图去刷一次 Buffer 页面，但具体刷多少，需要根据当前系统的负载来决定。如果 InnoDB 处于活动状态，则每次只刷一个比例的页面即可，以防给系统造成太大的压力，而如果 InnoDB 处于空闲状态，则每次都刷 100% 的页面。当然，这只是后台刷盘而已，刷盘还包括好几个时机，具体的将会在后面 Buffer 系统中有详细的讲述。

到此为止，InnoDB 的启动已经完成，可以看出，它做了很多的操作。从它的启动过程，也可以对 InnoDB 有一个更清晰的认识，可以知道作为一个强大的存储引擎，包括哪些模块、启动的流程是什么、启动过程不同阶段的意义等。而知道这些，对平时运维及学习 MySQL 是很有帮助的。

InnoDB 存储引擎的关闭

上面所讲的是 InnoDB 是如何启动的，日常所了解的启动之后的所有操作，或者运维的一些场景。但是，如果要关闭一个数据库，那么相对于 InnoDB 的打开操作，它是如何关闭的呢？这一节就主要来讲一下这个话题。

首先，可以看一下关闭操作的代码实现，如下。

```
innobase_end(
    handlerton*      hton, /*!< in/out: InnoDB handlerton */
    ha_panic_function type __attribute__((unused)))
    /*!< in: ha_panic() parameter */
)
{
    int err= 0;

    if (innodb_initiated) {
```

```

    srv_fast_shutdown = (uint) innobase_fast_shutdown;

    innodb_inited = 0;
    hash_table_free(innobase_open_tables);
    innobase_open_tables = NULL;
    if (innobase_shutdown_for_mysql() != DB_SUCCESS) {
        err = 1;
    }
    srv_free_paths_and_sizes();
    my_free(internal_innobase_data_file_path);
    mysql_mutex_destroy(&innobase_share_mutex);
    mysql_mutex_destroy(&commit_cond_m);
    mysql_cond_destroy(&commit_cond);
}

DEBUG_RETURN(err);
}

```

从代码中可以看出,首先要处理的就是我们所熟知的一个参数:innobase_fast_shutdown,对应的真正参数名为 innodb_fast_shutdown。

这个参数很明确,指的就是在关闭数据库时,如果设置为 0,则这种方式就是最保险的一种数据库关闭方式,它需要做的相关操作包括一个全量的回滚段 PURGE 操作、CHANGE BUFFER 的 merge 操作、将所有日志刷入日志文件及磁盘、将所有 Buffer Pool 中的脏页刷入到数据文件及磁盘,这些做完之后,才会去关闭数据库,这种方式是最慢的一种,也是最安全的,不过需要的时间可能比较长,一般用于正常维护数据库。

而如果设置为 1,就是我们默认设置的方式,这种方式相比上面设置为 0 的时候,不会去做 CHANGE BUFFER 的 merge 操作,也不会去做一个全量的 PURGE 操作,其他操作和上面是一样的,这种方式相对上一种而言,安全方面的操作都做了,只是为了在一定程度上加快一点速度,有两个操作没有做。所以,这是一般情况下要设置的一种方式。

而如果设置为 2,则上面所做的操作中,只会将已经产生的日志刷入到磁盘,其他所有操作都不会做。这样的话,产生的实际后果就相当于一次数据库异常退出,即所谓的 Crash。也许唯一比真正 Crash 好的就是,这种方式还有点时间用来尽可能地保证数据的安全,将没有刷入磁盘的日志数据刷到磁盘中。那么很明显,这个实例如果再重启的时候,就会做 Crash Recovery,因为 Buffer Pool 中的脏数据,肯定存在没有被写入到数据文件中的部分,所以还会通过日志来恢复,这个时间可能就会比较长。而如果日志文件个数比较多,并且文件比较大,那么这个恢复时间相应地就会变长。也就是说,这些时间其实都是需要的,只不过要看花在什么地方,要做什么操作。如果是为了快速转移数据,那么设置成这种方式是有必要

的，数据恢复可以放在新的机器上去做。也就是说，不同的参数只为不同的需求，而设置为1是一种正常的工作模式。

接下来，就是真正处理关闭存储引擎的函数——`innobase_shutdown_for_mysql`了，精简之后的代码如下。

```

/*****
/* Shuts down the InnoDB database.
   @return DB_SUCCESS or error code */
UNIV_INTERN
dberr_t
innobase_shutdown_for_mysql(void)
/*=====*/
{
    ulint i;

    /* 1. Flush the buffer pool to disk, write the current lsn to
       the tablespace header(s), and copy all log data to archive.
       The step 1 is the real InnoDB shutdown. The remaining steps 2 - ...
       just free data structures after the shutdown. */

    logs_empty_and_mark_files_at_shutdown();

    /* 2. Make all threads created by InnoDB to exit */
    srv_shutdown_state = SRV_SHUTDOWN_EXIT_THREADS;
    for (i = 0; i < 1000; i++) {
        if (!srv_read_only_mode) {
            /* a. Let the lock timeout thread exit */
            os_event_set(lock_sys->timeout_event);
            /* b. srv error monitor thread exits automatically,
               no need to do anything here */
            /* c. We wake the master thread so that it exits */
            srv_wake_master_thread();
            /* d. Wakeup purge threads. */
            srv_purge_wakeup();
        }

        /* e. Exit the i/o threads */
        os_aio_wake_all_threads_at_shutdown();
        /* ... */
    }
}

```

```

/* This must be disabled before closing the buffer pool
   and closing the data dictionary. */
btr_search_disable();
ibuf_close();
log_shutdown();
lock_sys_close();
trx_sys_file_format_close();
trx_sys_close();
dict_close();
btr_search_sys_free();

/* 3. Free all InnoDB's own mutexes and the os_fast_mutexes inside them */
os_aio_free();
que_close();
row_mysql_close();
srv_mon_free();
sync_close();
srv_free();
fil_close();

/* 4. Free the os_conc_mutex and all os_events and os_mutexes */
os_sync_free();

/* 5. Free all allocated memory */
pars_lexer_close();
log_mem_free();
buf_pool_free(srv_buf_pool_instances);
mem_close();
ut_free_all_mem();

return(DB_SUCCESS);
}
#endif /* !UNIV_HOTBACKUP */

```

从上面的代码中可以看到，存储引擎的关闭，共包括如下五个部分。

- 第一步，从注释中可以看到，`logs_empty_and_mark_files_at_shutdown` 函数所做的工作就是将所有 Buffer Pool 中的脏页刷入到磁盘，当然这个行为具体要不要做，还要根据上面所说的 `fast_shutdown` 参数而定，且要将最新的日志 LSN 写入到日志文件及 `ibdata` 的相应位置中，记录最新的日志情况。这一步是真正的 InnoDB 存储引擎的关闭，也是最费时间的一部分，当然也还是决定于 `fast_shutdown` 参数。做完了这一步，数据库的数据就在逻辑上及物理上保证了完整性。如果此时数据库崩溃了，也不会有什么问题。

- 第二步，就是通知并等待所有正在工作或是处于等待状态的线程退出，包括最著名的 Master 线程、PURGE 线程等，当然还有所有的异步 IO 线程。在线程退出之后，接下来就是关闭所有的子模块、子系统，包括 Change Buffer、日志系统、锁管理系统、事务系统及数据字典系统等。
- 第三步，就是释放一些子模块，将 InnoDB 所占用的一些资源释放掉，比如操作系统的 IO 队列、文件系统等。
- 第四步，只需要释放操作系统层面的用于多线程同步的临界区及事件等对象。
- 第五步，也是最后一步，就是将所有使用的内存都释放掉，比如内存大户 Buffer Pool、内存堆 heap 等。

完成上面这些操作，InnoDB 就属于正常关闭了。从这五步可以看得出来，最重要的就是第一步，涉及日志、检查点、事务、Buffer Pool 等模块，也是作为一个数据库最重要的一些子模块。关于这一步，再来看一下精简之后的代码段，并以注释的方式讲解其中所做的操作，其中保留了很多英文注释，以保证可以看到原汁原味的说明。

```

/*****
/* Makes a checkpoint at the latest lsn and writes it to first page of each
data file in the database, so that we know that the file spaces contain
all modifications up to that lsn. This can only be called at database
shutdown. This function also writes all log in log files to the log archive. */

UNIV_INTERN void logs_empty_and_mark_files_at_shutdown(void)
{
    /* local variables ... */

    /* 看到这个，应该会有有一种亲切感，所以还是留着 */
    ib_logf(IB_LOG_LEVEL_INFO, "Starting shutdown...");

    /* Wait until the master thread and all other operations are idle: our
algorithm only works if the server is idle at shutdown */
    srv_shutdown_state = SRV_SHUTDOWN_CLEANUP;
loop:
    os_thread_sleep(100000);
    /* 有兴趣的可以看一下这段代码，就像上一句英文注释所说，就是一堆
判断退出及等待操作。为了保证下面操作的安全，需要确保所有线程都
处于空闲状态。这里的循环条件就是判断是不是所有线程都是idle状态，
如果不是，则继续循环等待 */
    if (还有线程没有处于idle状态的)
        goto loop;
}

```

```
/* At this point only page_cleaner should be active. We wait
   here to let it complete the flushing of the buffer pools
   before proceeding further. */
```

```
srv_shutdown_state = SRV_SHUTDOWN_FLUSH_PHASE;
```

```
/* 一直等待page_cleaner退出 */
```

```
while (buf_page_cleaner_is_active) {
    ++count;
    os_thread_sleep(100000);
}
```

```
/* 从这里可以看到之前所介绍的fast_shutdown参数的使用之处。从下面的
   注释中也可以看出，设置为2的时候，相当于做一次宕机操作 */
```

```
if (srv_fast_shutdown == 2) {
    if (!srv_read_only_mode) {
        ib_logf(IB_LOG_LEVEL_INFO,
            "MySQL has requested a very fast shutdown "
            "without flushing the InnoDB buffer pool to "
            "data files. At the next mysqld startup "
            "InnoDB will do a crash recovery!");
```

/* 这段注释很重要，保留了源码中的说明，这里就是将所有已经产生的日志内容写入到日志文件及磁盘中，而不会刷Buffer脏页。需要注意的是，即便是要快速的关闭数据库，还是要把仅剩的并且必须要做的一些事情做了，那就是刷日志，因为如果不刷这些，就有可能丢数据，那样就真正宕机了。

```
In this fastest shutdown we do not flush the
buffer pool:
it is essentially a 'crash' of the InnoDB server.
Make sure that the log is all flushed to disk, so
that we can recover all committed transactions in
a crash recovery. We must not write the lsn stamps
to the data files, since at a startup InnoDB deduces
from the stamps if the previous shutdown was clean. */
```

```
log_buffer_flush_to_disk();
}
```

```
/* 正因为是快速关闭，所以这里就直接结束了这个函数的工作，下面
   所有的操作也就不会再做了，直接return返回。*/
```

```
srv_shutdown_state = SRV_SHUTDOWN_LAST_PHASE;
fil_close_all_files();
```

```

        return;
    }

    /* 到了这一步,说明数据库是正常关闭,不是快速关闭,所以做一次检查点。
       检查点的原理及工作在第11章会专门做详细讲解,这里只说他与日志、
       Buffer Pool相关,做了很多工作,从而保证了系统的一致性 */
    if (!srv_read_only_mode) {
        log_make_checkpoint_at(LSN_MAX, TRUE);
    }

    lsn = log_sys->lsn;

    /* 将文件刷盘,保证所有文件的写入都落在磁盘上 */
    if (!srv_read_only_mode) {
        fil_flush_file_spaces(FIL_TABLESPACE);
        fil_flush_file_spaces(FIL_LOG);
    }

    /* 将最新的LSN写入到ibdata文件相应的位置中去(具体什么位置,第11章会做介绍) */
    srv_shutdown_state = SRV_SHUTDOWN_LAST_PHASE;
    if (!srv_read_only_mode) {
        fil_write_flushed_lsn_to_data_files(lsn, 0);
        fil_flush_file_spaces(FIL_TABLESPACE);
    }

    /* 关闭所有文件 */
    fil_close_all_files();
    return;
}

```

到这里,InnoDB 存储引擎的关闭,就真正做完了。在函数 `innobase_end` 中看到的剩下的几行代码,已经不重要了,大可不必在意。

6 InnoDB 数据字典

背景

说到数据字典,这是任何一个通用型关系数据库所必备的,它是用来存储元数据信息的表,属于系统表。所有针对对象的操作,都会使用到系统表,对于了解知名大型通用数据库 (Oracle 或 SQL Server 等) 的同学而言,数据字典是很明确的,可以看到的。从中可以轻易得知目前数据库中有哪些对象、对象之间的关系等,也可以直接去查这些表来获取最新的元数据信息,非常方便,甚至有很多 Oracle 动态视图就是通过直接对数据字典的查询来实现的。但是这些同学在接触到 MySQL 之后,发现一切都改变了。第一次用的时候,可能就是想看看数据字典表的信息,但找遍了所有能找的地方,都没有找到相关的信息,给人的感觉就是 MySQL 是一个黑盒子。

当然,在了解 MySQL 数据库的同学看来,这不是很容易么?如果想看表结构,直接 `show create table tablename;` 就好了,想看到所有列的信息,直接使用 `show full columns from tablename;` 就可以了,应有尽有啊!但其他同学还是想看到数据字典的表,看到之后才踏实,那么这个时候,了解 MySQL 的同学可能就会说:“那这个确实没有,只能这样。”

但是,对于一个有兴趣学习 MySQL,想深入了解其设计原理及架构的同学而言,能看到数据字典是一件非常重要的事情。在 MySQL 中,可以说看不到、不支持这个功能,但不能说没有这个功能。看不到是有原因的,原因就是 MySQL 是一个插件式的数据库管理系统。

有数据字典的好处是,用户可以很明确简单地了解 MySQL 元数据管理的实现原理,比如创

建一个表时，对哪些字典表有影响、影响是什么、在内部是如何管理的等。如果看不到这些，可能很多人就会有疑惑。InnoDB 是如何管理这些元数据的，难道只有 .frm 文件？这个二进制文件的完整性如何保证？但实际上，这些是由看不到的系统表来保证的。frm 文件，对于 InnoDB 来说只是一个为了与 MySQL 兼容的附属品而已。如果从一开始就能了解到本节所要讲的内容，在学习及使用 InnoDB 时，就能少走一些弯路。

在 InnoDB 中，系统表实际上是看不到的，不像 Oracle 那样可以方便地通过一个查询语句就能得到其中的内容。原因就是，MySQL 是一个插件式的数据库管理系统。它的结构分两层，分别是 Server 层和存储引擎层，这两层需要相互配合才能一起友好地工作。但是，作为一个插件式的数据库管理系统，存储引擎层可以有多个存储引擎的插件，但 Server 层只有一个。这样的话，很明显，只能是存储引擎层来配合 Server 层了。因为一些历史原因导致每个存储引擎的实现程度并不相同，而且不同的存储引擎具有不同的功能，其各自的运用场景也不一样，这就使得存储引擎能实现的一些高级功能，在 Server 层会有不同体现，而 Server 层一般做不到对存储引擎区别对待，所以只能是一视同仁。

最早的默认存储引擎为 MyISAM，它是没有数据字典的，关于表结构，它所拥有的只有 .frm 文件，所以这导致了 InnoDB 也必须要有这个文件才能使得 Server 层识别并管理它。对于 Server 层来说，一个表是什么存储引擎，这是表的属性。具体深入到每一个存储引擎内部，数据字典表就不能被 Server 层来管理了，因为它是存储表的表，这就导致 InnoDB 的数据字典不能被用户感知了。想要了解这些知识，只能通过源代码，或者手册简单了解一下。

系统表结构

InnoDB 有四个最基本的系统表，用来存储用户定义的表、列、索引及索引列等信息，这些表分别为 SYS_TABLES、SYS_COLUMNS、SYS_INDEXES、SYS_FIELDS。下面分别对每一个表做简单介绍。

SYS_TABLES

用来存储所有以 InnoDB 为存储引擎的表，每条记录对应已经定义的一个表。

- NAME：表示一个表的表名。
- ID：表示这个表的 ID 号（8 个字节）。
- N_COLS：表示这个表的列的个数，建表指定的列数（4 个字节）。
- TYPE：表示这个表的存储类型，包括记录的格式、压缩等信息（4 个字节）。
- SPACE：表示这个表所在的表空间 ID 号（4 字节）。这个表对应的主键列为 NAME，同时还有一个在 ID 号上的唯一索引。

另外，由于历史原因，MIX_ID、MIX_LEN、CLUSTER_NAME 这三个位置暂时使用不到。

SYS_COLUMNS

用来存储 InnoDB 中定义的所有表中所有列的信息，每一列对应这个表中的一条记录。

- TABLE_ID：表示这个列所属的表的 ID 号（8 字节）。
- POS：表示这个列在表中是第几列（4 字节）。
- NAME：表示这个列的列名。
- MTYPE：表示这个列的主数据类型（4 字节）。
- PRTYPE：表示这个列的一些精确数据类型，它是一个组合值，包括 NULL 标志、是否有符号数的标志、是否是二进制字符串的标志及表示这个列是真的 VARCHAR（数据存储用两个字节）（4 字节）。
- LEN：表示这个列的数据长度，但不包括 VARCHAR 类型，因为这个类型在记录里面存储了数据长度（4 字节）。
- PREC：表示这个列数据的精度，但目前好像没有使用（4 字节）。这个表的主键列为（TABLE_ID、POS）。

SYS_INDEXES

用来存储 InnoDB 中所有表的索引信息，每条记录对应一个索引。

- TABLE_ID：表示这个索引所属的表的 ID 号（8 字节）。
- ID：表示这个索引的索引 ID 号（8 字节）。
- NAME：表示这个索引的索引名。
- N_FIELDS：表示这个索引包含的列个数（4 字节）。
- TYPE：表示这个索引的类型，包括聚簇索引、唯一索引、DICT_UNIVERSAL、DICT_IBUF（插入缓冲区 B+ 树）（4 字节）。
- SPACE：表示这个索引数据所在的表空间 ID 号（4 字节）。
- PAGE_NO：表示这个索引对应的 B+ 树的根页面（4 字节）。这个表的主键列为（TABLE_ID、ID）。

SYS_FIELDS

用来存储所有索引中定义的索引列，每一条记录对应一个索引列。

- INDEX_ID: 这个列所在的索引 (8 字节)。
- POS: 这个列在某个索引中是第几个索引列 (4 字节)。
- COL_NAME: 这个索引列的列名。这个表的主键列为 (INDEX_ID、POS)。

字典表加载

从上面的字典表就可以了解到 InnoDB 是如何管理表、索引、列及关键字的, 那么这几个表是如何加载的呢? 首先从建库说起。

在 InnoDB 启动的时候, 如果是新建数据库, 则需要初始化库, 所以首先需要做的就是创建字典管理的 B+ 树等信息。在第 5 章中, 已经了解到了函数 `innobase_start_or_create_for_mysql` 的功能, 其中调用了函数 `dict_create`, 就是用来创建数据字典的, 因为 InnoDB 中系统表的结构、个数等都是固定的, 不会有人修改, 所以在初始化库的时候只需要创建这几个表的存储 B+ 树 (与上面所说的索引对应, 一个索引一个 B+ 树) 即可, 同时将这几个 B+ 树的根页号存储在一个固定的位置, 就不需要将这几个表自身的信息存储在系统表中了。对于一个 B+ 树, 只要能找到其根页面, 就可以检索其数据了, 而数据库中系统表的数量是有限的, 所以这几个系统表对应的索引根页面被存储到一个指定位置, 按照一定的格式, 就可以找到这些系统表及操作它们的数据了, 也就可以把它们当作用户表一样去操作了, 不同的是, 系统表被藏在了 InnoDB 内部, 而用户表是暴露在用户面前的。

关于数据字典表根页面位置的存储方式, InnoDB 用了一个专门的页面 (0 号表空间 0 号文件的 7 号页面) 来管理字典信息, 这个页面用来存储上面提到的这四个表的五个根页面号 (有五个索引, `SYS_TABLES` 包含两个索引), 以及下一个表 ID 值 (全局变量, 创建新表时的 ID 号从这里分配, 每分配一个, 这个 ID 号要加 1)、下一个索引 ID 值 (创建索引时的 ID 号从这里分配, 每分配 1 个加 1)、下一个表空间 ID 值 (与上同理)、Rowid (这个是表中的 Rowid 号, 这个在下面另做介绍) 这四个值。

上面这些操作的实现都是通过函数 `dict_hdr_create` 完成的, 该函数完成的还有 `SYS_TABLES` 的两个索引、`SYS_COLUMNS` 的一个索引、`SYS_INDEXES` 的一个索引、`SYS_FIELDS` 的一个索引的创建, 创建索引是通过 `btr_create` 来完成的。

在 InnoDB 创建 B+ 树及一些其他初始化操作之后, 就通过函数 `dict_boot` 加载常驻内存的四个系统表并读取一些其他信息, 上面已经提过, 系统表的个数及结构都是不会被修改, 所以直接通过固定的硬编码构建这几个表即可, 比如: 创建 `SYS_TABLES`, 如下。

```
table = dict_mem_table_create("SYS_TABLES", DICT_HDR_SPACE, 8, 0);
dict_mem_table_add_col(table, heap, "NAME", DATA_BINARY, 0, 0);
dict_mem_table_add_col(table, heap, "ID", DATA_BINARY, 0, 0);
```

```
dict_mem_table_add_col(table, heap, "N_COLS", DATA_INT, 0, 4);
.....
```

创建 SYS_TABLES 表的索引, 如下。

```
index = dict_mem_index_create("SYS_TABLES", "CLUST_IND",
    DICT_HDR_SPACE,
    DICT_UNIQUE | DICT_CLUSTERED, 1);
dict_mem_index_add_field(index, "NAME", 0);
```

过程大致如上面所述。

初始化之后, 因为它是常驻内存的, 所以这四个表挂在一个全局的字典结构中。

```
struct dict_sys_struct{
    mutex_t      mutex;
    row_id_t     row_id;
    hash_table_t* table_hash;
    hash_table_t* table_id_hash;
    UT_LIST_BASE_NODE_T(dict_table_t) table_LRU; /*!< LRU list of tables */
    ulint        size;
    dict_table_t* sys_tables; /*!< SYS_TABLES table */
    dict_table_t* sys_columns; /*!< SYS_COLUMNS table */
    dict_table_t* sys_indexes; /*!< SYS_INDEXES table */
    dict_table_t* sys_fields; /*!< SYS_FIELDS table */
};
```

很明显可以看到, 结构体中最下面的四个就是用来存储对应的四个字典表的, 其中第一个成员 `mutex` 是一个临界区, 不做解释。成员 `row_id` 的管理会在本章最后一节“Rowid 管理”中专门讲述。

结构体中下面的链表及 HASH 表用来存储 InnoDB 中所有表的缓存, 包括系统表、用户创建的表, 这里有两个 HASH, 一个是按照名字缓存的, 一个是按照 ID 来缓存的, 各为其用罢了。LRU 链表用来管理表对象的缓存, 涉及淘汰操作。这部分内容, 就与第 4 章中所介绍的表对象缓存有关系了, 这里不做展开。

上面介绍了字典对象存储及管理, 下面来介绍一下普通用户表的一个加载过程。当用户访问一个表时, 系统首先会从表对象缓存池中查找这个表的 SHARE 对象, 如果找到了, 则直接从其实例化表对象空间链表中拿一个空闲的实例化的表对象出来使用, 如果没有一个可用的实例化对象, 则需要重新打开 (实例化这个表), 在实例化这个表的时候, 需要找到这个表的字典信息, 包括这个表本身、列信息及索引信息等。这个操作就通过下面介绍的主体 `dict_table_get_low` 来实现, 精简之后的代码如下。

```

dict_table_t*
dict_table_get_low(
/*=====*/
    const char* table_name) /*!< in: table name */
{
    dict_table_t* table;

    /* 先从表缓存中找这个表 */
    table = dict_table_check_if_in_cache_low(table_name);

    /* 如果找到了,但这个表已经损坏,则要在日志中报出一些异常来,并且提示
       可以通过参数innodb_force_load_corrupted来忽略这种异常,但要尽量避免
       这样的设置,因为只有在查问题时或一些特殊场景中才使用这个参数 */
    if (table && table->corrupted) {
        fprintf(stderr, "InnoDB: table");
        ut_print_name(stderr, NULL, TRUE, table->name);
        if (srv_load_corrupted) {
            fputs(" is corrupted, but"
                " innodb_force_load_corrupted is set\n", stderr);
        } else {
            fputs(" is corrupted\n", stderr);
            return(NULL);
        }
    }

    /* 如果在缓存中没有找到,就从数据字典中找找看,也就是需要查系统表了 */
    if (table == NULL) {
        table = dict_load_table(table_name, TRUE, DICT_ERR_IGNORE_NONE);
    }

    /* 最后不管找没找到这个表,都返回这个对象,但有可能是NULL值 */
    return(table);
}

```

从上面的代码中可以看到,首先从字典缓存中寻找,看这个表有没有被缓存在上面提到的 HASH 表中,如果找到则直接拿去使用即可,如果没有找到则通过 `dict_load_table` 从上面提到的四个系统表中加载。那么从字典表新加载一个表的过程是怎样的呢?看一下函数 `dict_load_table` 被精简之后的样子,如下。

```

UNIV_INTERN
dict_table_t*

```

```

dict_load_table(
    const char* name, /*!< in: table name in the databasename/tablename format */
    ibool      cached, /*!< in: TRUE=add to cache, FALSE=do not */
    dict_err_ignore_t ignore_err)
{
    /* local variables ... */
    heap = mem_heap_create(32000);

    /* 这个是一个很重要的东西，在第11章中，会做详尽的解释，此处可以忽略 */
    mtr_start(&mtr);

    /* 先找到系统表SYS_TABLES，根据这个表的表结构，拼一个记录出来。
       通过这条记录，去SYS_TABLES表中查找到想要的表 */
    sys_tables = dict_table_get_low("SYS_TABLES");
    sys_index = UT_LIST_GET_FIRST(sys_tables->indexes);

    /* 构造SYS_TABLES格式的元组 */
    tuple = dtuple_create(heap, 1);
    dfield = dtuple_get_nth_field(tuple, 0);

    /* 设置搜索列信息，因为主键是NAME列，所以直接通过这个列来搜索即可。
       找到唯一想要查找的表信息，如果没有找到，那就是没有这个表 */
    dfield_set_data(dfield, name, ut_strlen(name));
    dict_index_copy_types(tuple, sys_index, 1);

    /* 打开B+树，创建游标，对树进行搜索，搜索的模式是GE（大于等于） */
    btr_pcur_open_on_user_rec(sys_index, tuple, PAGE_CUR_GE,
                             BTR_SEARCH_LEAF, &pcur, &mtr);

    /* 搜索完成，取出游标找到的相应记录。当然，如果没有找到的话，记录就是一个特殊值 */
    rec = btr_pcur_get_rec(&pcur);

    /* 处理没有找到，或者这条记录已经被删除（打了删除标记）的情况，
       如果记录为特殊值，则关闭游标及B+树搜索，退出，并返回NULL值 */
    if (!btr_pcur_is_on_user_rec(&pcur)
        || rec_get_deleted_flag(rec, 0)) {
        /* Not found */
        btr_pcur_close(&pcur);
        mtr_commit(&mtr);
        mem_heap_free(heap);
    }
}

```

```

    return(NULL);
}

/* 解读找到的这条表记录, 创建一个table对象, 这就是被加载的
   最原始的表信息, 不包含列、索引、外键等信息 */
err_msg = dict_load_table_low(name, rec, &table);

innobase_format_name(table_name, sizeof(table_name), name, FALSE);

/* 对SYS_TABLES的搜索完成, 关闭游标, 执行mtr_commit */
btr_pcur_close(&pcur);
mtr_commit(&mtr);

/* 正如其名, 加载这个表的所有列, 和上面的方法是类似的,
   从SYS_COLUMNS表中加载所有关于想要查找的表的列信息 */
dict_load_columns(table, heap);

/* 如果需要, 将这些信息加入到字典缓存中去 */
if (cached) {
    dict_table_add_to_cache(table, TRUE, heap);
} else {
    dict_table_add_system_columns(table, heap);
}

/* 同理, 加载表中的所有索引信息 */
err = dict_load_indexes(table, heap, index_load_err);

/* 同理, 加载表中的所有外键信息 */
err = dict_load_foreigns(table->name, NULL, true, true, ignore_err);

/* 返回完整信息的表结构 */
return(table);
}

```

首先, 找到 SYS_TABLES, 过程与找普通表是一样的, 也是首先找缓存, 找不到则再从系统表中加载, 但这似乎是废话, 因为系统表在系统启动时就加载了, 不可能找不到。找到之后, 构造一个查询键值, 因为是通过名字查询的, 同时 SYS_TABLES 有一个按照名字排序的索引, 所以可以直接按照名字构造查询键值。然后, 从 B+ 树中查询对应记录, 如果找不到则报错, 找到之后, 根据这个表的记录格式解析这个记录, 取出 ID、N_COLS、TYPE、SPACE 这些基本信息, 然后根据这些信息创建一个表的内存对象。到这里, 这个表的自身对象就加载完成了, 接着开始加载其所有列信息。

加载列操作与加载表原理基本一致，对应系统表为 SYS_COLUMNS，它的聚簇索引列为 TABLE_ID 和 POS。索引查找时，扫描记录中的 TABLE_ID 如果是相同的，则 POS 是从小到大排序的，所以构造查询某一个表的所有列的键值时只需要通过 TABLE_ID 来查询即可。扫描完成之后按照顺序取出所有列信息并一一构造内存对象，这样这个表的所有列也相应加载完成。

加载列的时候，还有一点需要注意，在 InnoDB 中，一个表的列包括两部分，一部分是用户创建表时指定的列，另一部分则是系统列，包括 Rowid、TRXID 及 ROLLPTR 三个列。Rowid 表示记录的行号；TRXID 表示这条记录最后一次被修改的事务号，主要用于事务的多版本（MVCC）管理；ROLLPTR 也是用来实现多版本的，如果一条记录被某一个用户修改之后，另一个用户在这条记录不可见时，查询这条记录要找到其原来的值，那么 ROLLPTR 指定的就是其原来值的位置，这个位置其实就是在修改时写下的回滚记录位置。所以，对于任何一个表，其中所包括的列除了用户定义的列之外，还包括这三列。

接下来是加载这个表的索引信息，加载索引是从 SYS_INDEXES 中查询的，原理与上面的 SYS_COLUMNS 一样。这个表的关键字是 TABLE_ID 及索引 ID，所以具有相同 TABLE_ID 的所有索引记录都是按照索引 ID 排序的。每一条记录对应一个索引，需要加载 ID、名字、N_FIELDS、TYPE、PAGENO、SPACE 等基本信息。

对于索引的加载，还需要加载它对应的所有关键字信息，这些信息存储在 SYS_FIELDS 系统表中，这个表的关键字是 INDEX_ID、POS，所以一个索引的所有关键字列都是按照 POS 排序的。这点很重要，因为如果有多个排序列的话，顺序不同，排序结果也是不同的，所以一定要按照 POS 的值从小到大加载（B+ 树存储顺序），索引的关键字信息包括索引 ID 号、POS、列名，一个索引加载了所有具有指定索引 ID 号的关键字列后，一个索引的加载即完成。但是加载关键字还有一点需要注意，如果一个索引不是唯一索引，则需要将表中已经加载的 Rowid 列以这个索引第一个关键字列的身份加载到这个索引中；如果是唯一索引，则不需要加载 Rowid 列，而是直接加载自身定义的列。在加载完所有的关键字列后（要么是 Rowid 列，要么是自定义列），还需要加载另外两个系统列，包括 TRXID 及 ROLLPTR 两个列。而对于聚簇索引，因为这个索引存储了表中所有的列，所以后面还需要加载除关键字之外的所有列，这些列是按照建表时的顺序加载的，而对于二级索引，这些列是不需要加载的。按照相同道理，将一个表中所有的索引加载完成。

上面已经叙述了所有 InnoDB 中系统表及索引管理的内容，但对于系统表，不能像一般想象的普通表那样，用户并不能查询其中的内容，只能是系统内部自己管理及维护，比如 SYS_TABLES 等这些表，MySQL 是不承认的。这个问题也与 MySQL 的特性有关，因为它是插件式的，必须兼容所有的存储引擎，但并不是每个存储引擎都有这些系统表，所以，用户是不能通过 MySQL 来直接查询或者访问 InnoDB 系统表的。

另外一个问题需要重点强调一下，我们在上面讲述时，重点讲述了表、列、索引等信息，但在表对象的上一层，即库信息，有没有相应的数据字典表呢？这里我想说的是，没有。

上面介绍的函数 `dict_load_table` 中，第一个参数是用来指定表名的，对这个参数的注释说明是这样的：`/* !< in: table name in the databasename/tablename format */`，可以看到，这个名字传入的时候，就是以格式 `databasename/tablename` 传入的，并且在构造搜索记录时，直接用了这个参数。也就是说，一个表所属的库信息，在 InnoDB 的字典表中是存储于 `SYS_TABLES` 表中 `name` 列的，并没有额外的表来存储库信息。所以，可能有很多人想知道的 InnoDB 中表与库之间关系的维系方式，其实就是在这里。

当然，这是从数据字典的方面考虑的。如果是 MyISAM 情况下，没有数据字典怎么办？这个问题就很明确了，MyISAM 压根就没有这些信息，也没有对应关系，或者说唯一的对应关系就是它被放到哪一个目录名下了，它所在的目录名，即为所在的数据库名。也正因为这样，一个 MyISAM 数据库表，可以通过 COPY 元数据文件和数据文件来迁移。MyISAM 是完全独立的，但 InnoDB 就做不到了，因为 InnoDB 表的相关信息还存储在 `ibdata` 中。

也正因为这些原因，如果有想要重命名一个数据库的需求，可行的办法有下面两种。

- 如果是 MyISAM，就直接在离线情况下，在文件系统的层面，直接重命名所在的目录名，即可实现这个需求；而对于 InnoDB 存储引擎，即使是离线状态下，也不能通过这种方式来改库名，因为库名存储在了数据字典中，需要在线修改。
- 如果要在在线的情况下（这种方式，所有存储引擎都是可以的），实现这个需求，那就需要通过一个迂回的方式来做了，取出一个库中所有的表，然后拼一大堆语句出来，首先创建一个新的库，比如 `newdb`，而老的库名为 `olddb`，接着就是执行很多 `rename table olddb.table to newdb.table`。将所有表都执行这样的操作之后，老库下面就没有表了，而新库下面就包括了所有的表，这样就变相地实现了重命名库的需求。

Rowid 管理

在 InnoDB 中，用户表中的记录不一定会都会有一个 Rowid 列，Rowid 只有在一个表没有定义主键时，也就是需要 Rowid 作为聚簇索引列的时候才会被分配给这个表。而 Rowid 的管理分配，并不是一个表独享一个 ID 空间，而是全局的，所有表都共享这个 ID 号。

一般情况下，或是像上面的 `TABLEID`、`INDEXID` 一样，每次分配一个就更新一次字典页面的值，但对于 Rowid 并不是这样，因为插入操作比创建一个表或者索引的操作频繁多了，每次都去修改未免对效率影响太大，所以 InnoDB 也做了相应的优化，就是每分配一个 Rowid，系统只是在内存中加 1，不会修改页面，只有当这个值为 256 的倍数时才会写入一次。那么我们自然会想到，如果插入 200 次，这些值还没有被写入，这时系统重启了，ID 号岂不是会重复使用？这当然不是问题，因为数据库启动时，调用的函数 `dict_boot` 中还做一工作，

就是将上次写入的 Rowid 值向上对齐 256 后再加上 256，这样就不会有问题了，大不了可能会因跳过很多 ID 号导致这个值增长太快而已。

总结

关于 MySQL 的数据字典，目前还有些问题需要改进，而目前也正在做。InnoDB 对于 MySQL 来说，会越来越重要，从目前的发展就可以看得出来，两者之间的关系变得越来越紧密，如此一来，Server 层和引擎层（这里专指 InnoDB）的合作就越生硬，会从以前 InnoDB 很“委屈”地去配合 Server 层，变为 Server 层和 InnoDB 相互依赖、相互配合，两层之间的交互变得越来越多，很明显可以看出来，Server 层现在对 InnoDB 有了特别的“照顾”，也可以说，MySQL 和 InnoDB 现在已经进入了一个蜜月期。

像这样发展下去，它们联合起来之后，体现在功能上就是，一方（InnoDB）的强大由于另一方（MySQL Server）的局限而不能体现出来的局面越来越少，而整体的表现就是 MySQL 会越来越强大。拿数据字典管理来说，有可能出现的改进就是数据字典功能更强大，可以包含更丰富的信息，随之而来，表的功能和配合也会更丰富，并且可以做到 DDL 的回滚功能，从而在熟知的 ALTER TABLE 方面，有更加友好的功能及更安全的在线改表方案。

另一个值得期待的功能就是 InnoDB 的数据字典信息可以在 MySQL 层面看得到、摸得着，从而给运维带来一种新的思路，新的展现方式，而不是通过一个危险的 information_schema 来统计库表相关信息。有人可能说，这可能实现吗？谁知道呢，实现了不是更好吗？

InnoDB 数据存储结构

在第 1 章中，已经介绍了 MySQL 数据库系统涉及的文件组织架构，如图 7.1 所示。

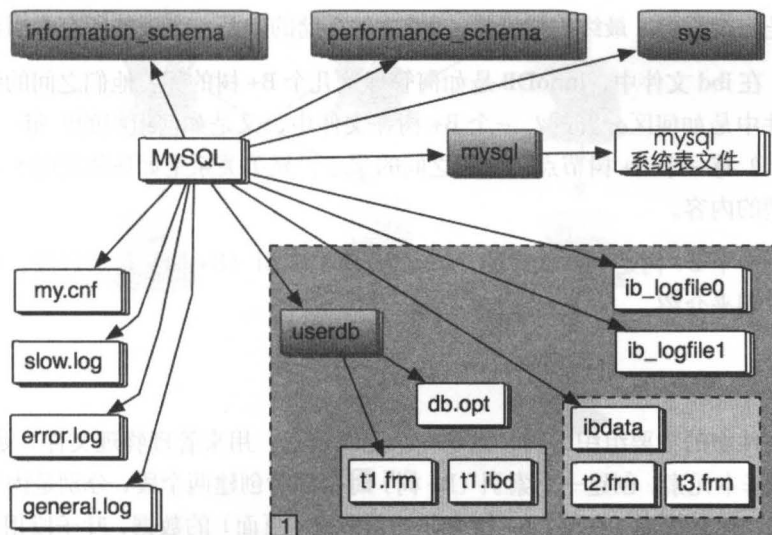


图 7.1

图 7.1 中所示的多个文件，一起组成了一个 MySQL 数据库系统，有的相互依赖，有的彼此独立，有的可有可无，有的必不可少，每一个文件的存在，都为 MySQL 添砖加瓦，组成一个完整的数据库生态环境。但这是相对于整个数据库系统来说的，如果只考虑某一个文件，

他们在文件内部的管理也都有各自的方法,管理的好坏,直接影响数据库的运行效率。这一章,就专门介绍一下使用 InnoDB 存储引擎时,一个表空间文件的管理方法。

表空间文件组成结构

InnoDB 存储引擎在存储设计上模仿了 Oracle 的存储结构,其数据是按照表空间进行管理的。新建一个数据库时,InnoDB 存储引擎会初始化一个名为 `ibdata1` 的表空间文件,默认情况下,这个文件会存储所有表的数据,以及我们所熟知但看不到的系统表 `SYS_TABLES`、`SYS_COLUMNS`、`SYS_INDEXES`、`SYS_FIELDS` 等。此外,还会存储用来保证数据完整性的回滚段数据,当然这部分数据在新版本的 MySQL 中,已经可以通过参数来设置回滚段的存储位置了。

InnoDB 存储引擎的设计很灵活,可以通过参数 `InnoDB_file_per_table` 来设置,使得每一个表都对应一个自己的独立表空间文件,而不是存储到公共的 `ibdata1` 文件中。独立的表空间文件只存储对应表的 B+ 树数据、索引和插入缓冲等信息,其余信息还是存储在默认表空间中。

以一个单表文件(表空间文件)为切入点,来介绍这种文件的管理方式。现在已经知道,这个文件所存储的内容主要就是 B+ 树(索引),一个表可以有多个索引,也就是在一个文件中,可以存储多个索引,而如果一个表没有索引的话,用来存储数据的被称为聚簇索引,也就是说这也是一个索引。最终的结论是,ibd 文件存储的就是一个表的所有索引数据。

那问题来了,在 ibd 文件中,InnoDB 是如何管理这几个 B+ 树的呢?他们之间的组织关系是什么?在文件中是如何区分开的?一个 B+ 树在文件中,又是如何组织的?每一个 B+ 树管理单位是什么?每一个 B+ 树节点,他们之间的父子、兄弟关系是如何体现的?这些正是这节想要说清楚的内容。

首先介绍一下一个 B+ 树是如何构成的。InnoDB 中的索引(B+ 树)是通过段、簇、页面构成的,下面分别来介绍。

段

段是表空间文件中的主要组织结构,它是一个逻辑概念,用来管理物理文件,是构成索引、表、回滚段的基本元素。创建一个索引(B+ 树)时会同时创建两个段,分别是内节点段和叶子段,内节点段用来管理(存储)B+ 树中非叶子节点(页面)的数据,叶子段用来管理(存储)B+ 树中叶子节点的数据。也就是说,在索引数据量一直增长的过程中,所有新的存储空间的申请,都是从“段”这个逻辑概念中申请的,在内节点分裂时申请新节点就从内节点段中申请,在叶子节点分裂申请新节点时就要从叶子段中申请了。一个索引,包括两个段,那么一个表的段的数目,就是索引的个数乘以 2 了。更形象的解释是,ibd 文件,就是由多个段组成的,没有任何其他空间是脱离了段的管理的。

页面

现在已经知道段和簇的关系了，但簇的物理空间内部还需要继续被切分并高效管理。我们平常津津乐道的“页面”就是簇在细分之后的产物，它是簇的组成单位，也是段所管理的最小单位、数据库文件管理的最小单位，当然也是文件中空间分配的最小单位。

一个簇中可以包括多个页面（默认为 64 个页面），这个页面数通常被叫做“簇的大小”。这些页面都归这个簇管理，在逻辑上（页面号都是从小到大连续的）及物理上都是连续的。在向表中插入数据时，如果一个页面已经被写完，系统会从当前簇中分配一个新的空闲页面出来使用，如果当前簇中的 64 个页面都被分配完，系统会从当前页面所在段中分配一个新的簇，然后再从这个簇中分配一个新的页面来使用，依此类推。更简单地说，表空间文件就是被划分成相等长度的块，每一个块就是一个页面，一个页面默认为 16KB，上面已经说过，一个文件中没有任何空间是脱离了段的管理而存在的。那么，这里可以说，对于一个文件，没有任何空间不是以页面的形式而存在的。

图 7.3 是段、簇、页面之间的一个简单关系图。

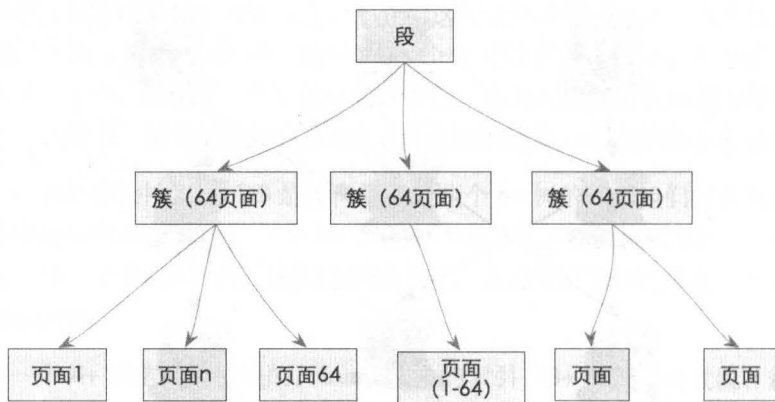


图 7.3

段、簇、页面组织结构

上面已经提到，段是由多个簇组成，簇是由默认的 64 个页面组成的，但在具体实现上，它们是如何组织、如何实现的呢？下面就主要从 InnoDB 的源代码的角度来讨论它们的组织方式，同时讲述在组织这三层结构时，InnoDB 是如何做到可扩展、高效的页面管理的。

一个表空间可以有多个文件，每个文件都有各自的编号，创建一个表空间时，至少有一个文件，这个文件被称为“0 号文件”。上面已经提到，一个文件是被切割为等长（默认 16KB）的块，这个块通常被称为页面，那么在“0 号文件”的第一个页面（page_no 为 0）中，存储了这

个表空间中所有段簇页管理的入口，那么在这个页面中，可存储的数据就是 16KB，但通常都会有页面头信息会占用一些空间，真正的管理信息数据是从页面偏移为 FIL_PAGE_DATA (38) 的位置开始的，这个位置存储了表空间描述信息，描述信息中包括如下主要内容。

- FSP_SPACE_ID: 当前表空间 ID 号，每一个表空间都有一个唯一的 ID 号，在创建时分配。在第 6 章中可以了解到，这个 ID 的分配会在 ibdata 文件的第 0 号文件的第 7 个页面中存储和管理，每分配一个新的表空间，这个值会加 1，并将最新值写入这个页面相应的位置中。
- FSP_SIZE: 当前表空间总的页面数，即一个表空间可以有多个文件。这个值表示表空间中所有文件按照页面（默认 16KB）大小划分的页面数之和，在文件空间不足需要扩展时，这个值会被更新为扩展后的大小。
- FSP_FREE: 表空间所有段中的簇都是由表空间统一管理的，这个地址是一个链表头指针，表空间中所有的空闲簇都以链表的形式存储在这个链表中。上面已经提到，表空间中空间扩展的最小单位就是簇，页面是属于簇的，所以，文件中所有游离出现的都是一个一个的簇。新分配的或已经清空数据的簇，都会被挂在这个链表中。这个链表，相当于是一个表空间的簇池子，不用了的簇可以放到这里，某一个段要分配新的簇时就从这里取。
- FSP_FREE_FRAG: 一个簇可以管理很多页面（默认值 64 个页），如果这个簇中已经有被使用（分配）的页面，这个簇就被称为半满簇，这个地址就是存储所有半满簇的链表头指针。这个链表和上面的 FSP_FREE 是类似的，只不过存储的簇的性质不同，FSP_FREE 存储的是完全空闲的簇，而这个是只有部分页面是空闲的簇。
- FSP_FULL_FRAG: 如果一个簇中的所有页面都已经被分配（使用）了，那么这个簇就被称为满簇，这个地址就是用来存储所有满簇的链表头指针。
- FSP_FRAG_N_USED: 这个值表示上面的 FSP_FREE_FRAG 链表中，所有已经被使用过的页面数。在分配页面时，每从 FSP_FREE_FRAG 链表中分配一个空闲页，这个值都会加 1 并写入到文件中。
- FSP_SEG_ID: 在表空间中，每一个段都有一个唯一分配的 ID 号，这个值表示的是下一个段的 ID 号，每次使用之后这个值都会自加，以保证所有的 ID 号都是不相同的。
- FSP_SEG_INODES_FULL: 这里需要先介绍一下 Inode，Inode 是用来管理段的，简单来说，一个 Inode 就代表一个数据段。Inode 可以说是一个结构体，它也是像上面一样，按顺序存储到 Inode 页面中，一个 Inode 页面可以存储多个 Inode 节点。如果页面中所有的空间都用来存放已经使用的 Inode，则这个页面就称为满 Inode 页面，否则称为半满 Inode 页面。FSP_SEG_INODES_FULL 就是用来存储所有的满 Inode 页面的链表头指针。
- FSP_SEG_INODES_FREE: 用来存储所有半满 Inode 页面的链表头指针，或者是空闲的 Inode 页面。

Inode 节点用来管理一个段，一个 Inode 中包括以下内容。

- FSEG_ID: 表示这个段的 ID 号，在创建时唯一分配。
- FSEG_NOT_FULL: 一个段管理很多簇，这些簇都是属于这个段的，这个地址用来存储所有半满簇的链表头指针。
- FSEG_NOT_FULL_N_USED: 这个地址用来存储上面半满簇链表中所有已经使用的页面总数。
- FSEG_FREE: 这个地址用来存储所有空闲簇的链表头指针。
- FSEG_FULL: 这个地址用来存储所有满簇的链表头指针。

从上面的叙述中可以知道，表空间控制信息中有满簇链表、半满簇链表、空闲簇链表，而段的 Inode 信息中也有这些信息，但这两个其实是不同的。表空间中的链表管理的是整个表空间中所有的簇，包括满簇、半满簇及空闲簇，而段的 Inode 信息中管理的是属于自己段中的满簇、半满簇及空闲簇。当段新申请一个簇时，如果段上面没有空闲的簇，此时它就会从表空间的簇链表中找，找到后从相应链表中摘下来挂到段空闲簇链表中。

上面一直提到的簇，是一个段的组成元素，段通过三个链表将不同状态的簇管理起来，链表都是双向链表。在段控制信息中，链表头分别是 FSEG_FREE、FSEG_FULL、FSEG_NOT_FULL，链表节点就是簇，每个簇上面都有向前指针和向后指针。每一个簇通过一个簇描述符来表示，簇是实际的物理存储空间，簇描述符是一个结构体，其内容如下。

- XDES_ID: 这个值表示这个簇所属段的 ID 号，对应的值就是上面 Inode 结构中的 FSEG_ID 值。
- XDES_FLST_NODE: 这个地址用来存储簇的链表指针，包括向前指针和向后指针，每一个指针（地址）都是一个页面地址，包括 page、boffset。page 表示这个簇描述符在文件中的哪一个页面，它是一个页面号；boffset 表示簇描述符在 page 页面中的偏移地址，表示从这个位置开始就是这个簇的描述符地址。这里需要注意的是，当前所讲述的内容是一个簇描述符，簇描述符管理的是一个簇的状态及页面的使用情况等，簇链表就是多个这样的簇描述符通过 XDES_FLST_NODE 向前向后的指定而被链接起来的表，所以，这个指针指向的，其实还是这样的结构。一个簇描述符，是把这些信息存储在一个页面中，所以只要找到一个簇描述符的首地址，也就是存储 XDES_ID 的位置，就可以找到一个簇描述符了。这个链表指针的内容包括 page、boffset 两部分，page 表示的是这个簇描述符所在的页面号，而 boffset 表示的是簇描述符所在的页面 page 中首地址的偏移量。这里，只需要把一个簇的描述符理解为一个簇即可，因为簇是被簇描述符管理的，所以这里可以等同理解。至于簇描述符是如何与簇页面对应上的，下面会有解释。
- XDES_STATE: 表示当前这个簇的状态，状态包括 XDES_FREE（空闲簇）、XDES_FREE_FRAG（半满簇）、XDES_FULL_FRAG（满簇）、XDES_FSEG（属于一个段）。

- XDES_BITMAP: 这个地址存储的是一个位图, InnoDB 通过这个来管理一个簇中所有页面的使用情况。每一个页面用两个位来表示, 簇大小 FSP_EXTENT_SIZE 表示的是一个簇的页面数, 默认值为 64, 所以 XDES_BITMAP 占用的总长度为 $64 \times 2 / 8 = 16$ 个字节。用两个位来表示一个页面, 第一个位表示这个页面是否使用, 第二个位现在还没有使用, 所以一个簇描述符的大小为 $XDES_ID(8) + XDES_FLST_NODE(12) + XDES_STATE(4) + 16(XDES_BITMAP \text{ 的大小}) = 40$ 个字节。

从上面得知, 一个簇描述符, 占用 40B 的空间, 这些信息是被存储在页面中的, 这个页面, 就被称为簇描述页, 它所存储的内容就叫作簇描述符。

在 InnoDB 中, 一个簇描述页面默认要管理 16384 个页面, 簇大小默认为 (FSP_EXTENT_SIZE) 64 个, 而一个簇描述符的大小为 40B, 所以一个簇描述页面中可以描述的簇的个数为 $(UNIV_PAGE_SIZE - \text{页面头长度}) / 40$, 其中 $UNIV_PAGE_SIZE = 16384B$ 表示页面大小。但一般情况下, 都不会将簇描述页面存储满, 因为在一个表空间中, 簇描述页面的存储是每隔 16384 个页面就有一个是簇描述页面, 所以簇描述页面中只需要描述 16384 个页面即可。每个簇大小为 64 个页面, 所以需要存储的簇描述符个数为 $16384 / 64 = 256$ 个。由此可知, 页面大小为 16384 个, 其实只用了 $256 \times 40 = 10240B$ 的空间, 其他空间都是空闲的, 那这样的话, 256 个簇描述符存储在一个簇描述页面中, 可以管理 16384 个页面, 也就是 256 个簇, 可以通过图 7.4 来更形象地了解它们的关系。

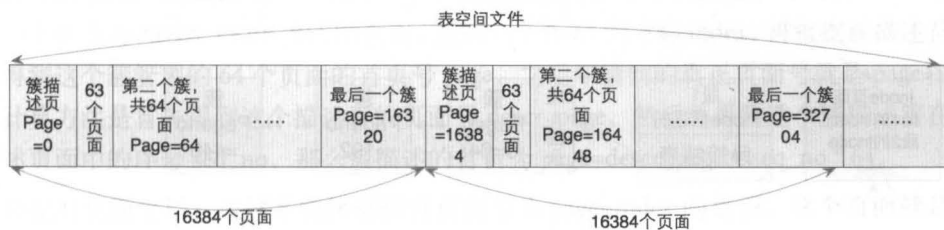


图 7.4

从图 7.4 中可以看出, 一个表空间文件以簇为单位被分隔开, 同时每 16384 (默认值) 个页面又用一个簇描述页面来描述, 在每 16384 个页面的分隔中, 第一个页面都用来做簇描述页面, 用来描述后面 $16384 - 64 = 16320$ 个页面。第一个簇 (其实是第一个页面) 只是用来做簇描述页面的, 它没有被加入到表空间的簇链表中, 也没有被加入到段的簇链表中, 第一个簇后面的 63 个页面是被空出来的, 并没有被使用, 其实是被浪费掉的空间, 真正的空间使用是从第二个簇开始直到最后一个簇, 所以簇描述页面中实际上只有 255 个簇描述符。

上面在解释 XDES_FLST_NODE 的时候, 提出了一个问题, 就是簇描述页面中存储的簇描述符上的信息没有体现它管理的页面是哪些, 只有一个位图用来表示管理的 64 个页面是否已经被使用, 但是从图 7.4 中可以看到, 每 16384 个页面就有一个簇描述页, 所以第一个描述

页的页号为 0，第二个为 16384，依此类推。同时，一个描述页面中，所有描述符描述的页面都是相隔 64 连续，所以只要知道一个描述符的位置，就可以计算出这个描述符所管理的页面号范围，因此在描述符中是不需要存储它的页面信息的，通过描述符的地址即可得到。

计算方法概括如下，根据描述符的地址得到描述页的页面号 `describe_page_no`，然后再根据簇描述符在簇描述页面中的偏移量得到它在这个簇描述页中的序号 `index`，管理的范围 `scope` 可以通过下面的公式计算出来。

```
min_scope = describe_page_no + index * FSP_EXTENT_SIZE。
max_scope = describe_page_no + (index + 1)* FSP_EXTENT_SIZE。
min_scope <= scope < max_scope
```

很明显，`max_scope - min_scope` 的值就是 `FSP_EXTENT_SIZE`，即 64。

到现在为止，大概的结构已经有了，现在来看一下整体的组织结构图，如图 7.5 所示。

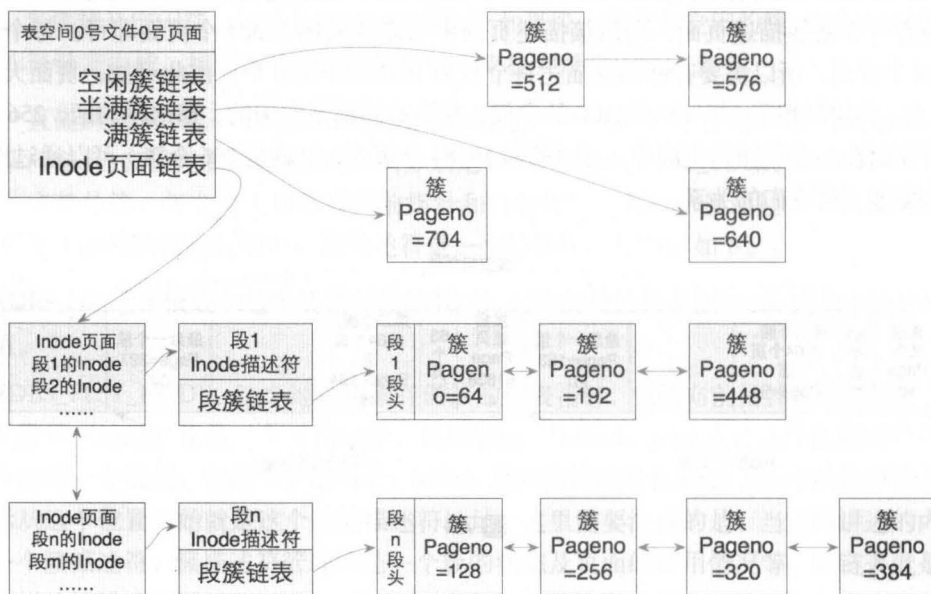


图 7.5

图 7.5 描述了表空间、Inode 页面、Inode、段、簇、页面之间的关系，也是 InnoDB 文件系统管理架构图。

图 7.5 中所表示的最小管理单位是簇，每个簇中都包含 64 个页面，每个簇都是以首页面 `Pageno`（页面号）表示，这些簇的首页面可以是连续的（相隔 64），也可以不连续（相差大于 64，是 64 的整数倍）。

下面简单叙述一下创建一个段的过程，用来说明文件管理的实现过程。

- 根据表空间 ID 号得到表空间头信息。
- 从得到的表空间头中分配一个 Inode，首先判断 FSP_SEG_INODES_FREE 链表中是否还有空闲的 Inode 页面，如果有，则从页面的数据存储位置开始扫描，每一个 Inode 的大小是固定的，所以扫描的步长也是固定的。每找到一个 Inode 后判断 Inode 描述符中的 FSEG_ID 是否为 0，如果是，则没有使用，否则便是已经使用过了，找到第一个为 0 的则返回，说明已经找到了合适的 Inode，如果找到后发现这个 Inode 是这一页的最后一个 Inode，则将这个页面从 FSP_SEG_INODES_FREE 链表中摘下来，同时将这个页面插入到 FSP_SEG_INODES_FULL 链表中。如果 FSP_SEG_INODES_FREE 链表中没有空闲的 Inode 页面，则需要重新分配一个 Inode 页面，分配后将所有的 Inode 描述符中的 FSEG_ID 置为 0，表示未使用，然后将这个页面链接到 FSP_SEG_INODES_FREE 链表中，然后直接从这个页面中分配一个空闲的 Inode，过程如上所述。
- 给新分配的 Inode 指定 SEG_ID 号，这个 ID 号要从表空间头的 FSP_SEG_ID 中取出来作为新段的 ID 号，然后将这个 ID 号写入到 Inode 的 FSEG_ID 中，同时更新 FSP_SEG_ID 中的值，更新为 ID+1，表示下一个段的 ID 号。
- 初始化这个 Inode 信息，将偏移 FSEG_NOT_FULL_N_USED 处的值置为 0。初始化链表 FSEG_FREE、FSEG_NOT_FULL 和 FSEG_FULL。
- 从这个段中分配出一个页面，分配页面时首先找到表空间头上的半满簇链表 FSP_FREE_FRAG，然后从链表中找一个簇描述符，找到簇描述符之后，从它的 XDES_BITMAP 中找一个状态为 XDES_FREE_BIT 的页面，返回一个 0~63 的下标 index，再根据簇描述符计算得到这个簇管理的 64 个页面的首页号 page，之后申请到的真正页面号就是 page+index。计算方法是首先得到这个描述页的页面号 descr_page，然后再得到这个簇描述符在簇描述页面中的序号 seq_no，那么簇描述的首页为 page=descr_page + seq_no * 64。
- 分配好页面之后，通过系统缓存得到页面号为 page+index 的页面，这个页面就是这个段的首页面。在一个段的首页上，需要记录这个段对应的 Inode 的位置，Inode 的位置存储在页面头中，分别是 FSEG_HDR_OFFSET (INODE 在 INODE 页面中的偏移)、FSEG_HDR_PAGE_NO (INODE 所在的 INODE 页面号)、FSEG_HDR_SPACE (INODE 所在的表空间号)。
- 到此为止，一个段就分配完成了。以后如果需要在这个段中分配空间，只要找到其首页，然后找到对应的 INODE 即可分配空间。

上面的实现过程相对应的 InnoDB 源码实现，精简之后如下。



```
buf_block_t*
fseg_create_general(
    ulint space, /* space id */
```

```

    uint page, /* page where the segment header placed */
    uint byte_offset, /* byte offset of the created segment header on the page */
    mtr_t* mtr /* in/out: mini-transaction */
)
{
    /* local variables... */
    /* 正如上面参数中所描述的page一样, 如果传入值不为0, 则说明
       外面调用时, 需要将段头信息放到所传入的页面中, 而对应的
       byte_offset参数就是要存储在这个页面中的偏移量, 而如果
       page为0, 则说明是要创建一个独立的段, 并且将这个段本身的段头
       信息存储到自己的第一个页面中去。关于page=0, 下面有相应的处理 */
    if (page != 0) {
        block = buf_page_get(space, zip_size, page, RW_X_LATCH, mtr);
        header = byte_offset + buf_block_get_frame(block);
    }

    /* step 1 @above */
    space_header = fsp_get_space_header(space, zip_size, mtr);

    /* step 2 @above */
    inode = fsp_alloc_seg_inode(space_header, mtr);
    if (inode == NULL) {
        goto funct_exit;
    }

    /* Read the next segment id from space header and increment the
       value in space header */
    /* step 3 @above */
    seg_id = mach_read_from_8(space_header + FSP_SEG_ID);
    mlog_write_ull(space_header + FSP_SEG_ID, seg_id + 1, mtr);

    /* step 4 @above */
    mlog_write_ull(inode + FSEG_ID, seg_id, mtr);
    mlog_write_ulint(inode + FSEG_NOT_FULL_N_USED, 0, MLOG_4BYTES, mtr);
    flst_init(inode + FSEG_FREE, mtr);
    flst_init(inode + FSEG_NOT_FULL, mtr);
    flst_init(inode + FSEG_FULL, mtr);

    mlog_write_ulint(inode + FSEG_MAGIC_N, FSEG_MAGIC_N_VALUE, MLOG_4BYTES, mtr);
    for (i=0; i < FSEG_FRAG_ARR_N_SLOTS; i++) {
        fseg_set_nth_frag_page_no(inode, i, FIL_NULL, mtr);
    }
}

```

```

    }

    /* step 5 @above */
    if (page == 0) {
        block = fseg_alloc_free_page_low(space, zip_size,
                                         inode, 0, FSP_UP, mtr, mtr);
        if (block == NULL) {
            fsp_free_seg_inode(space, zip_size, inode, mtr);
            goto funct_exit;
        }

        header = byte_offset + buf_block_get_frame(block);
        mlog_write_ulint(buf_block_get_frame(block) + FIL_PAGE_TYPE,
                        FIL_PAGE_TYPE_SYS, MLOG_2BYTES, mtr);
    }

    /* step 6 @above */
    mlog_write_ulint(header + FSEG_HDR_OFFSET,
                    page_offset(inode), MLOG_2BYTES, mtr);
    mlog_write_ulint(header + FSEG_HDR_PAGE_NO,
                    page_get_page_no(page_align(inode)),
                    MLOG_4BYTES, mtr);

    mlog_write_ulint(header + FSEG_HDR_SPACE, space, MLOG_4BYTES, mtr);

funct_exit:
    return(block);
}

```

从代码中,可以更加清楚地看到一个段的实现方式,这样应该比只看到一堆文字更加踏实吧?至此,就已经熟知了 InnoDB 的文件管理方式。其实核心目的,就是为了更好地管理一颗 B+ 树,或者说是一个索引。那么,现在来看一下,在已经知道段簇页管理方式,以及如何创建一个段的情况下,InnoDB 是如何创建一个 B+ 树的,继续来看精简之后的代码,如下。

```

/* Creates the root node for a new index tree. */
UNIV_INTERN
ulint
btr_create(
    ulint    type, /* in: type of the index */
    ulint    space, /* in: space where created */
    ulint    zip_size, /* in: compressed page size in bytes or 0 for uncompressed

```

```

        pages */
index_id_t index_id,/* in: index id */
dict_index_t* index, /* in: index */
mtr_t*      mtr) /* in: mini-transaction handle */
{

    /* local variables... */

    /* Create the two new segments (one, in the case of an ibuf tree) for
    the index tree; the segment headers are put on the allocated root page
    (for an ibuf tree, not in the root, but on a separate ibuf header page) */

    /* 首先, 创建一个段出来, 这个函数实际上就是上面所介绍的fseg_create_general。
    从参数上可以知道, 这个段的地址, 要被存储到PAGE_HEADER + PAGE_BTR_SEG_TOP
    位置中, 并且传入的page参数为0, 说明存储到的页面就是这个段的首页 */
    block = fseg_create(space, 0, PAGE_HEADER + PAGE_BTR_SEG_TOP, mtr);
    if (block == NULL) {
        return(FIL_NULL);
    }

    page_no = buf_block_get_page_no(block);
    frame = buf_block_get_frame(block);

    /* It is a non-ibuf tree: create a file segment for leaf pages */
    /* 在创建首页之后, 再在这个页面的PAGE_HEADER + PAGE_BTR_SEG_LEAF
    位置创建一个段。很明显, 这个段就是一个B+树中的叶子段, 从名字
    PAGE_BTR_SEG_LEAF上也可以看得出来, 创建完成之后, B+树对应的两个
    段就被成功创建了 */
    if (!fseg_create(space, page_no,
        PAGE_HEADER + PAGE_BTR_SEG_LEAF, mtr)) {
        /* Not enough space for new segment, free root
        segment before return. */
        btr_free_root(space, zip_size, page_no, mtr);

        return(FIL_NULL);
    }

    /* Create a new index page on the allocated segment page */
    page_zip = buf_block_get_page_zip(block);

    /* 将上面所创建的段中的首页面初始化, 将其设置为一个索引页面

```



```

    的格式, 因为不同用途的页面, 其内部格式是不同的, 这里将要用作
    索引, 所以将其设置为索引格式 */
/* 另外需要注意的是, 这个页面, 就是所熟知的B+树根页面,
   也就是在第6章中介绍数据字典时说过的, 在数据字典
   的索引表中存储的根页面号 */
if (page_zip) {
    page = page_create_zip(block, index, 0, 0, mtr);
} else {
    page = page_create(block, mtr,
        dict_table_is_comp(index->table));
}

/* Set the index id of the page */
/* 设置这个根页中的索引ID, 也就是这个B+树属于哪一个索引 */
btr_page_set_index_id(page, page_zip, index_id, mtr);

/* Set the next node and previous node fields
   初始化页面链表信息, 因为目前只有一个页面, 前后都是FIL_NULL */
btr_page_set_next(page, page_zip, FIL_NULL, mtr);
btr_page_set_prev(page, page_zip, FIL_NULL, mtr);

/* We reset the free bits for the page to allow creation of several
   trees in the same mtr, otherwise the latch on a bitmap page would
   prevent it because of the latching order */

if (!(type & DICT_CLUSTERED)) {
    ibuf_reset_free_bits(block);
}

/* In the following assertion we test that two records of maximum
   allowed size fit on the root page: this fact is needed to ensure
   correctness of split algorithms */

ut_ad(page_get_max_insert_size(page, 2) > 2 * BTR_PAGE_MAX_REC_SIZE);

/* 返回根页面号, 在后面会将这个页面号写入到SYS_INDEXES表中属于这个
   索引那行记录的最后一列中, 也就是PAGE_NO列, 这样就可以通过一个索引
   找到它的根页面信息了 */
return(page_no);
}

```

这个函数所完成的工作就是创建一颗 B+ 树，也就是一个索引，当然这里所说的索引只是概念上的，还没有真正的数据，数据还需要等创建完成再将对应的数据写入到 B+ 树之后，才可以被称为一个真正的索引。所谓的 btr，在 InnoDB 中，表示的就是 B+ 树的处理，不管以它为开头的是函数，还是文件，都是相关的处理，是 B tree 的简称。

至此，所知道的关于数据字典、B+ 树、索引、段、簇、页等概念都被串联起来了。通过对代码的解释，这些内容变得更加有血有肉，不过在代码中还涉及一些其他内容，似乎是比较陌生的，比如 mtr、buf 等信息，这里没有做更详细的讲解，但第 11 章会讲到这些内容。

InnoDB 索引实现原理



背景

对于从事数据库行业的技术人员来说，B 树是必须要掌握的知识。在我研究生时期，一位在国内足够权威的数据库老师曾经说过：“我想不通为什么在《数据库原理》这门数据库必修课的教学大纲中，竟然没有将 B 树这章列入其中，这是多么荒唐的事情！”从这句话也可以看出，B 树对于数据库而言，是多么重要的一门技术。

或者换一个概念，大家所熟悉的——索引，这是日常工作必须要打交道的一个概念。我们所了解的大多数数据库，其索引都是通过 B 树或类 B 树实现的，今天的主题 InnoDB，正是索引中的一员，所以从这个角度来看，索引和 B 树，就可以等同起来。不过，更加精确地讲，InnoDB 是使用 B+ 树来实现其索引功能的。具体到 B+ 树和 B 树的区别，如上所言，这是应该在大学期间学习的内容，如果真的没学过，也没太大关系，此时还有机会，继续往下看。

B+ 树及 B 树的区别

首先，关于什么是 B 树，或者 B+ 树，在教科书或网上有很多相关介绍，可以直接获取，但它们之间的区别，一般都是理论方面的，从数据库应用的角度去解释可能更加容易理解。经总结，这两种数据结构的不同之处有如下五点。

- B 树中的同一键值不会出现多次，它有可能出现在叶子节点上，也有可能出现在内节点上；而 B+ 树的键一定会出现在叶子节点上，同时也有可能在非叶子节点中重复出现。简

单来说, B+ 树的内节点存储的都是键值, 键值对应的具体数据都存储在叶子节点上。

- 由于 B 树的每一个节点都存储了真实的数据, 会导致每一个节点存储的数据量变小, 所以整个 B 树的层数就会相对变高, 当数据量变大之后, 维护代价是比较大的, 而且层数越高, 搜索或修改的性能就会越低; 而在 B+ 树的内节点中, 只存储键值, 相对而言, 一个内节点存储的记录个数比 B 树多很多。由于 B+ 树是横向扩展的, 所以随着其中数据量的增长, 最终会成长为一个矮胖子, 不像 B 树一样是纵向扩展, 最终只会变成一个瘦高个子。这样整体而言, B+ 树在搜索时, 从上到下直到叶子节点只需要遍历层数个节点而已, 因此性能会比较高。
- B 树的查询效率与键在 B 树中的位置有关, (在叶子节点的时候) 最大时间复杂度与 B+ 树相同, 最小时间复杂度为 1 (在根节点的时候); 而 B+ 树的复杂度对某个建成的树是固定的。
- B 树中, 键的位置不固定, 且在整个树结构中只出现一次, 虽然可以节省存储空间, 但却使得插入、删除等操作复杂度明显增加。而且性能不平衡, 有可能会很快找到合适的位置, 也有可能需要做比较多的 IO 操作才能找到。而 B+ 树相对来说是一种较好的折中, 因为内节点相对叶子节点而言, 相当于是一个索引, 在插入的过程中, 只需要通过在每一层搜索一个节点, 依次找到叶子节点之后, 在叶子节点处做插入操作即可, 只是在遇到一个节点存储满了的情况下会进行 B+ 树分裂, 但总体而言性能还是比较稳定。
- B 树中, 所有的数据都只存储一份; 而 B+ 树中, 除了存储了所有数据的叶子节点外, 还有只存储键值数据的内节点, 所以, 在占用空间量方面, B+ 树比 B 树会多占用一些空间, 这部分空间就是 B+ 树内节点的所有空间, 但 B+ 树通过这种方式提高了整体性能, 更适合于性能要求很高的文件检索。

索引的设计

数据库是用来存储数据的工具, 存进去, 是为了更方便地取出来, 而且越快越好, 这样对性能的要求就非常高了。在计算机上运行一个任务, 一般有三部分涉及性能, 分别是内存大小、CPU 及磁盘的速度, 而索引是一种存储方式, 与它相关的最重要的部分就是磁盘, 所以磁盘性能的高低, 直接影响了在数据库中查找数据的效率。

磁盘的性能与读写顺序有关, 对于普通的机械硬盘, 顺序读写会比随机读写快很多, 这里涉及的机械硬盘的存储原理在很多相关书籍中都有介绍, 此处不再赘述。在加快数据库中数据的读写速度时, 需要尽可能地避免随机读写, 也就是说尽可能地读取连续的数据, 这样性能自然就会好一些。

除了硬盘读写顺序的影响之外, 还需要考虑读写操作本身的效率, 因为在读写时, 有些数据是必须要操作的, 也就是真正需要的那部分数据, 这部分数据被称为有效数据, 这部分数据

之外被同时读写的数据，被称为无效数据。索引的设计，必须要尽可能地降低无效数据的读写访问。

关系型数据库的结构有如下三个特点。

- 数据都是以行为单位一行一行存放的，一行中包括一个表（聚簇索引）或者一个索引（二级索引）中定义的所有列，多行数据可以连续一起存储。
- 一行数据中，一般都会有一个键，以及其他附属的列，可以称之为值，可以简单理解为一行数据就是一个键值对。有些人可能会有一个疑问，如果不定义主键怎么办。没关系，InnoDB 已经替你想到了这一点，它会在内部加上一个主键，即我们所熟知的 RowID 值，这样就有了一个默认的键，相应表中所有用户定义的列就是值了，也照样形成了一个键值对。
- 在键值对中，键值可以排序，还可以组合键值。

综合以上三个特点，以及 B+ 树的特点，这哥俩感觉就是天生一对，所以设计存储方式如下。

- 将磁盘空间或存储文件划分为许多大小相同的块（Block）或者页（Page），而在一个块中，可以存储多个数据行，多个数据行在一个块内的存储格式可以先不用考虑，这样设计就迎合了磁盘顺序读取性能比较高的特性，因为读取一条数据时，也很有可能读取其周边的数据，这对于相对固定的块来说，一次顺序 IO 就可以读取很多数据出来，在性能提高上是非常适合的，而如果不通过块来管理行的话，行为单位的管理就会非常碎，IO 也会非常随机，性能就变得差了。
- 在一个块内，所有数据行的组织管理也是需要讲究的。因为数据有可能经常会变，并且它的大小是相对固定的，有可能会存储满，所以内部是通过链表或数组的方式来管理的。
- 前面已经提到，每行数据就是一个键值对，并且键可以排序，在一个块内，所有的行数据也可以有序，这样利用经典的二分查找算法就可以很快地根据指定的键来找到对应的键值对数据或一定范围内的很多数据。
- 一个块的问题解决之后，如何形成一棵 B+ 树呢？现在很容易想到，可以让一个块作为一个 B+ 树的节点，这样通过块来承载数据，通过 B+ 树这个数据结构来组织不同块之间的关系，最终形成一个矮矮胖胖的 B+ 树结构。
- 因为行是一个键值对，而 B+ 树的特点是通过在内节点中只存储键来提高搜索性能，这两个特性正好匹配。很自然地，在 B+ 树中，内节点存储了行数据中的键，而叶子节点存储所有的行数据。通过内节点的键值及一个位置信息，内节点与下层节点或叶子节点之间的指针，就可以找到其孩子节点了。

现在，这个 B+ 树（索引）的雏形就出来了，再结合第 7 章所讲的 InnoDB 文件管理方法，就可以串联起来了。这里所说的块，其实就是一个页面，B+ 树所用到的块，就是被一个段，或

簇所管理的。

聚簇索引和二级索引

在查询数据时，一般都会在经常被查询的字段上建立一个索引，这正是利用了索引中被排序的键值。通过内节点的索引功能及叶子节点中数据的有序性，利用二分查找的方法，极大地提高了查找的性能，所以索引在数据库中的作用是至关重要的。

在之前的章节中介绍表空间文件管理时已经提到，一个表可以建立多个索引，但每一个表都有一个索引是存储了所有数据的，这个索引一般被称为“聚簇索引 (Clustered Index)”。聚簇索引在一个表中只有一个，且是建立在主键上面的，这个主键所包含的列可以是被隐藏的 Rowid 列，也可以是自增列，还可以是明确定义的不含 NULL 值的组合列等。除了聚簇索引之外的所有索引，都被称为二级索引 (Secondary Index，又称为辅助索引)，那么很明显，二级索引可以有多个，并且一般没有上限，想建多少都可以。不过，如果两个索引建立在同样的列，或者列组合上面，那么这两个索引就被称为重复索引或冗余索引，这在 MySQL 中一般都会以一个警告给出，通过 `show warnings` 可以看到如下的警告信息。

```
mysql> show warnings\G
```

```
***** 1. row *****
```

```
Level: Note
```

```
Code: 1831
```

```
Message: Duplicate index 'its3' defined on the table 'local.ts'. This is deprecated  
and will be disallowed in a future release.
```

```
1 row in set (0.00 sec)
```

从上面的特性我们可以知道，在一个表中，聚簇索引占用的空间肯定是最大的，因为它是存储了全部数据的，而二级索引是建立在某几个需要经常查询的列上面的，除了这几个列之外，剩下的就是用来“回表”的指针信息了，所以相对而言，二级索引的占用空间都会比聚簇索引小很多，特别是在一个表的列数很多或是这些列中包含大字段的情况下，因为我们一般都不会在大字段上直接建立索引。那这样比较下来，在我们统计一个表总的精确行数时（查 `COUNT*`），一些优化器就会选择表中最小的索引来作为统计的目标索引，因为它占用空间最小，IO 也会最小，性能相应更快一些。

上面说到了“回表”，所谓回表，就是在使用二级索引时，因为二级索引只存储了部分数据，如果根据键值查找到的数据不能包括全部目标数据，就需要通过二级索引的指针，也就是键值对中的值，来找到聚簇索引的全部数据，然后根据完整的数据取出所需要的列的过程。这种在二级索引中不能找到所有需要的数据列的现象，被称为非覆盖索引，反之称为覆盖索引。因为回表本身是需要去另一个索引（聚簇索引）中查找数据的，性能必然会受到影响，那为了尽可能地提高性能就需要尽量减少回表次数，所以可以试着将出现频率非常高的语

句中所有使用到的列以合适的顺序建一个二级索引，这样所有需要的列都被这个二级索引覆盖了，就不需要回表了，从而在一定程度上提高了性能。这虽然是一个好的做法，但需要去权衡，因为需要考虑语句中涉及的列数、这条语句出现的频率及这个索引最终的大小。最坏的情况是建一个和聚簇索引差不多大的二级索引，这样一方面是占用空间比较大，另一方面是维护这个二级索引会对这个表的整体修改性能造成一定的影响。所以，需要权衡各个方面，然后再决定要怎样做。

上面还说到，在统计总行数的时候，可以直接使用二级索引来做，是因为有一个很明显且很重要的前提：每个二级索引与聚簇索引的总行数是一样的，并且是一对一的关系。只不过在每一个索引中，数据行的排列顺序不同，可以想象二级索引行与聚簇索引行之间都有虚线相连，并且二级索引中的每一行都有且只有一条虚线指向聚簇索引中的一行数据，而聚簇索引的每一行，都会有相同个数的虚线指进来，这个数目就是二级索引的个数。至于二级索引与聚簇索引究竟是如何对应起来的，后面会进行详细讲述。

关于二级索引的个数，虽然没有限制，但可以想象一下，任何事物都是有两面性的：建一个索引，是为了提高性能，但这是以降低写入性能为代价的。因为所有索引，在表需要写入数据时，都需要去维护索引数据以保证所有索引都是最新、最准确的，所以可想而知，索引越多，写入性能也就越差，对索引上锁的时间也会越长。更严重的是，如果有唯一索引，为了保证唯一这个特性，每次修改都会去检查唯一性，在RR隔离级别下，经常会造成死锁，所以在建索引时一定要仔细权衡，建出来的索引要个个为精，个个有用，这样才能保证在最大程度提高性能的情况下，最小程度地影响对表的修改。

二级索引指针

现在已经知道，聚簇索引存储了所有数据，二级索引只存储了部分数据，但二级索引是为了提高性能建立的，所以经常会被使用到。那如果二级索引中的数据不能满足需求怎么办？这就用到了上面提到的“回表”，二级索引中每行记录中的指针也就发挥了作用。

关于聚簇索引及二级索引列之间的逻辑关系，分类如下。

- 自定义主键的聚簇索引。

索引结构：[主键列][TRXID][ROLLPTR][其他建表创建的非主键列]。

参与记录比较的列：主键列。

内节点 Key 列：[主键列]+PageNo 指针。

- 未定义主键的聚簇索引。

索引结构：[ROWID][TRXID][ROLLPTR][其他建表创建的非主键列]。

参与记录比较的列：只 ROWID 一列而已。

内节点 Key 列：[ROWID]+PageNo 指针。

- 自定义主键的二级唯一索引。
索引结构: [唯一索引列][主键列]。
参与记录比较的列: [唯一索引列][主键列]。
内节点 Key 列: [唯一索引列]+PageNo 指针。
- 自定义主键的二级非唯一索引。
索引结构: [非唯一索引列][主键列]。
参与记录比较的列: [非唯一索引列][主键列]。
内节点 Key 列: [非唯一索引列][主键列]+PageNo 指针。
- 未定义主键的二级唯一索引。
索引结构: [唯一索引列][ROWID]。参与记录比较的列: [唯一索引列][ROWID]。
内节点 Key 列: [唯一索引列]+PageNo 指针。
- 未定义主键的二级非唯一索引。
索引结构: [非唯一索引列][ROWID]。参与记录比较的列: [非唯一索引列][ROWID]。
内节点 Key 列: [非唯一索引列][ROWID]+PageNo 指针。

以上六种情况讲清楚了聚簇索引记录包含的列、二级索引记录包含的列,以及在非叶子节点中分别包含的列。因为索引是用来检索数据的,所以也解释了用来检查记录时,在二级索引及聚簇索引中,参与比较记录大小的列分别是什么,以及唯一索引与非唯一索引的区别等。



注意:上面讲述的索引列的顺序关系,与实际索引中记录的物理存储不是一回事,记录的存储格式是记录的格式,而这个是索引在内存中元组的组织关系,元组的顺序所体现的就是每个索引自己的逻辑顺序,以哪一列建的索引,哪一列就会在最前面起到优先排序的作用。

这里特别关注一下二级唯一索引的元组逻辑顺序。在二级唯一索引中,作为索引本身的索引列,就是上面所说的“键”,当这个元组需要回表时,在元组中存储的聚簇索引列信息,就是我们所说的“值”,这两者组合起来就形成了键值对。而对于二级非唯一索引而言,因为只有索引列本身再加上主键列才能保证索引记录是唯一的,所以这二者合起来才能构成我们所说的“键”,而“值”就为空了,也就是说,二级非唯一索引中,在记录构成方面,非叶子节点只是比叶子节点多了一个 PageNo 指针信息。

从上面可以看到,在二级索引元组中,首先存储的就是每个索引定义的索引列,接着就是这条记录对应的聚簇索引的主键列的值,而主键列是唯一的,所以二级索引回表时对应的记录也是唯一的,这样就形成了一种指针的效果。

不过有一点需要注意一下,二级索引回表时对应的聚簇索引,如果是用户自定义的,有可能是自增列,也有可能是有逻辑意义的单列或者组合列的聚簇索引;如果用户没有自定义,

则 InnoDB 会自动给聚簇索引分配一个主键列，不过是隐藏的列，即我们所熟知的 Rowid 列。基于此，如果是用户自定义的聚簇索引，则二级索引指针指向的就是聚簇索引所包含的列；如果没有自定义主键，那该指针就指向 Rowid 列。

神奇的 B+ 树网络

现在已经知道聚簇索引及二级索引中存储的内容了，但这仅局限于一条记录，或者是一个页面内的存储管理，而这些页面是如何构成一颗 B+ 树的呢？这是本节需要讲清楚的内容。

先看一个 B+ 树的图，如图 8.1 所示。

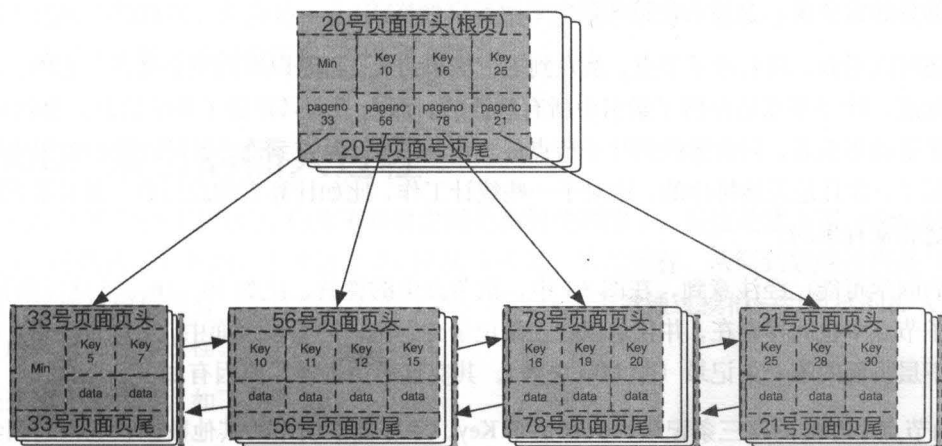


图 8.1

图 8.1 所示的就是一个具有两层节点的 B+ 树，也是我们俗称的索引，是一个已经成型并且内容基本完善的树形结构。在图 8.1 中，所有页面都已经被随机编号，所以可以认定所编的号码就是页面号，其中根页面号为第 20 号。

很明显，在两层树形结构中，除了 20 号根页面之外，其他节点（页面）都是叶子节点，或者说只有根页面才是内节点了。

从图 8.1 中可以看到，一条记录都被分成了两个部分，对于内节点而言（这里指的只是根页面了），一条记录所包含的内容，实质上是只有 Key 的，但在存储的时候，InnoDB 还是把 pageno 这个指针信息，以列的形式存进去了，只是在实际使用时并不将其计算在内，所以并不影响索引查询，这也可以理解为一个键值对的存储。而对于叶子节点而言，键值对的内容就发生了变化，因为叶子节点中的记录是不会再有指针指向其他页面的，并且它是存储了索引中完整数据的，这个键值对就包含了索引列的值（Key）以及索引中的其他列。当然这里还要区分聚簇索引和二级索引，如果是聚簇索引，那 data 部分存储的就是除主键列之

外的其他所有列组合，而如果是二级索引，则这里存储的就是这行记录对应的主键列组合，用于回表。

在每一层的最左边节点页面的最左边位置，都有一个 Min 记录，这是为了能很好地组织树形结构的指针，InnoDB 发明的一个虚拟记录。它和其他内节点中的记录一样，也是两部分，第一部分就是一个标记，表明它是最小记录 Min 即可，而第二部分就是一个 `pageno` 指针，指向下一层中最左边的记录，也就是用来指向存储比本页面中所有 Key 都小的记录页面。而对于其他节点中的 Min 记录，就是用来判断当前搜索是否已经到了一个页面的边界处的。

每一个页面都有一个页面头及页面尾，这两部分用来管理及标记页面状态、页面中的数据如何存储、有没有空闲空间、以什么顺序存储，以及如何解析并找到想要的的数据等。所以，这两部分非常重要，之后会在第 8 章中来解释页面格式。

另外还可以看到，所有叶子节点，从左到右，从小到大，都是以双向链表链在一起的。我们已经知道，叶子节点是存储了索引中所有数据的，而内节点只存储了 Key 信息，也就是说，如果想要遍历全表，只需要找到叶子节点最左边的节点，从左到右，就可以取出索引中所有的数据了，并且是天然排序的，这对于一些统计工作，比如计算表的总行数，或者是查询全表，是非常有用的。

细心的同学可能已经注意到，在图 8.1 中，根节点中的键值，比如 10、16、25 这三条记录，在叶子节点中也同样存在，并且是在这三条记录分别指向的下层页面中。更神奇的是，它们都是下层页面的第一条记录（除 Min 之外）。其实这不用奇怪，原因有如下三点。

- 内节点中存储的这三条记录，只存储了 Key，不存储索引中的其他数据，它只起到索引的作用，除此之外，别无他用。
- 叶子节点中，除了第 1 条记录在上层中有重复存储，其他记录不会有这样的现象，如果说浪费空间的话，浪费不会太多；
- 这正是前面所讲的 B+ 树的特点，为了提高性能，可以损失一些存储空间。可以想象一下，这样的树形结构，如果达到了四层的话会是什么样的效果。

不妨来算一算，假设每一个内节点页面可以存储 1000 条 Key，那如果将这棵树存满的话，总数据存储量可以像如下这样计算。

- 第一层根页面，只有一个页面，存储的 Key 是 1000 个；
- 第二层内节点，上层 1000 个 Key，这层就是 1001 个页面，相应的 Key 个数为 $1000 * 1001$ ；
- 第三层内节点，上层是 $1000 * 1001$ 个 Key，这层就是 $1000 * 1001 + 1$ 个页面，相应的 Key 个数为 $(1000 * 1001 + 1) * 1000$ ；
- 第四层为叶子节点，上层是 $(1000 * 1001 + 1) * 1000$ 个 Key，这层就是 $(1000 * 1001 + 1) * 1000 + 1$ 个页面。而叶子节点可以存储的数据量就不一定了，肯定不会达到 1000 个，假

设是 256 个，则整个树存满之后，可以存储的数据量就是 $((1000 * 1001 + 1) * 1000 + 1) * 256 = 256256256256$ 条记录，这可是千亿级别的记录啊！这么大数据量的 MySQL 单表存储应该没太有机会处理过。

通过计算，已经发现，一个只有 4 层的 B+ 树，可以存储这么巨大的数据量，每层之间都是相差千倍，这棵树要有多矮，多胖啊！简单算了一下，如果把每条记录想象为 1mm 的正方的话，把这些记录连起来，可以绕地球旋转 6.4 圈，太神奇了！

现在应该清楚了吧，这就是 B+ 树的魅力！如果想要在千亿级别的表中查找一个记录的话，只需要 4 个页面的 IO 即可完成最终数据的定位，在叶子节点中，只需要做一次内存级的二分查找即可找到具体的数据记录，这效率高得惊人。同时可以想象一下，如果改为使用 B 树存储千亿级别的数据，那会是一个什么样的景象，毫无疑问，IO 数量已经完全不是一个量级了，这里就不做过多讨论了。

InnoDB 索引的插入过程

现在已经知道 B+ 树的组织结构及不同层之间是如何关联的了，但这是建立在一个已经创建好的 B+ 树的认识上面的，至于这个 B+ 树从小到大、从无到有、从简到繁的过程是什么样子的，可能目前还没有一个比较深入的了解，那么这一节就通过模拟一组数据的插入过程，来了解索引的建立过程。

首先来做一些假设，如下。

- 每个页面（包括内节点和叶子节点）最多可以插入三条记录，插入第四条的时候，就会导致分裂。
- 插入数据为键值对，但我们只关注键，值可以不用关注，就简单地以 data 代替即可。
- 插入的数据序列为：10, 20, 5, 8, 23, 22, 50, 21, 53, 40, 9。
- 为了简单明了一些，Key 就是一个简单的 INT 类型的数字。
- 假设根节点页面号为 100。

现在来做第一次插入过程。此时，索引中还没有数据，所以这个 B+ 树只有一个空的根节点，因为一个页面只能存储三个 Key，首先将 10, 20, 5 插入进去，然后在页面内做数据排序，所以这 3 个 Key 插入之后，B+ 树应该是如图 8.2 所示的样子。

根据假设，根页面现在已经插入满了，但还有很多数据需要插入，此时如果再继续插入，则需要将根页面分裂（根节点的分裂），分裂过程如下。

1. 首先，创建一个新的叶子节点，假设申请出来的页面是 101 号。这里为什么创建一个新的叶子节点是有讲究的，因为我们已经知道，B+ 树的内节点与叶子节点实际上是通过

不同的段来组织的, 所以什么时候应该从什么段中取节点是需要注意的。这里从叶子节点取, 是因为现在的根节点同时还是叶子节点, 里面存储的数据都是全部的数据, 而不只是 Key, 所以此时需要叶子节点这么一个角色来将根节点的数据拿过来, 让其成为真正的属于内节点的根节点。这里还有一点需要注意的是, 在分裂过程中, 根节点始终是不会变的, 不管变成多大的树, 根节点的页面号始终如一。



图 8.2

2. 然后, 将原根页面的全部记录复制到新页面中, 原根页面的最小虚记录要指向新叶子节点, 同时将原根页面中的记录全部删除。
3. 最后, 将根页面的 Min 记录指针指向新的叶子节点 100 号页面, 这样就构成一个 B+ 树形结构了。

这样变换之后, 新的 B+ 树结构如图 8.3 所示。

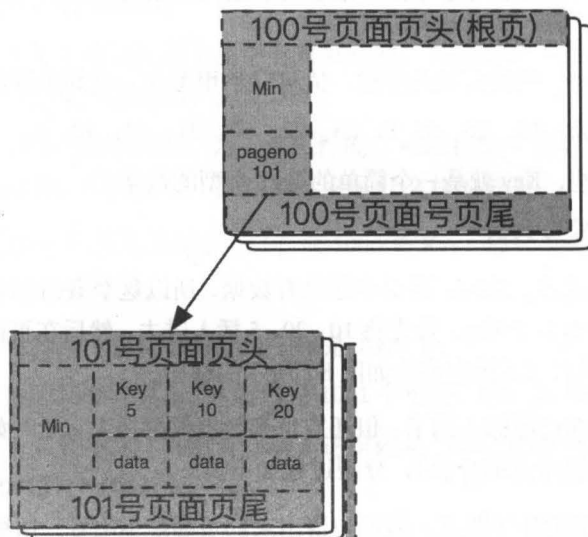


图 8.3

而此时根节点分裂是完成了，但是我们要插入的 Key——8 还没有插入进去，所以还需要继续插入，重新插入 8 的时候，可以想到，通过定位直接会找到第 101 号页面，在这个页面内插入时，发现还是没有空间，而此时这个页面是属于叶子节点的，所以这就涉及了又一次分裂——叶子节点的分裂，步骤如下。

1. 首先，要再创建一个新的叶子节点，假设页面号为 102。
2. 将 101 号页面的一部分数据移到 102 号页面中，这里的一部分一般是指一半，这里可以假设每次移过去 1 条。
3. 101 号页面和 102 号页面这都是叶子节点，一般称为兄弟关系，它们需要组成双向链表。
4. 将一半数据移到 102 号页面之后，102 号页面的数据就只有 Key 为 20 这条记录（注意这里是叶子节点，是完整记录），那么此时这个页面就需要与根节点挂上关系，需要做的就是将 20 这条记录的 Key 取出来，然后再加上一个指针信息，这里就是 102 号页面，组成一条新的记录插入到根页面中，那么至此这条记录就算指向了对应的儿子节点 102 了。

这样，叶子节点分裂并重新调整之后的 B+ 树如图 8.4 所示。

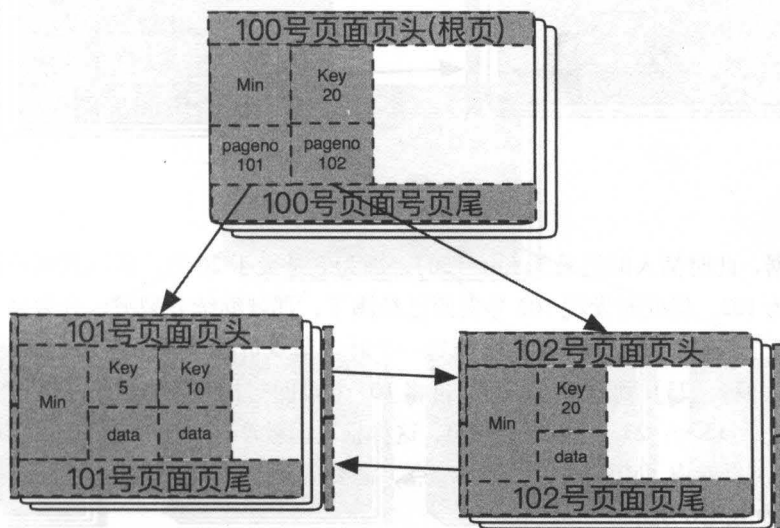


图 8.4

但是，可能有同学还在问，Key 为 8 这条记录怎么还没有插入进去呢？是的，上面都在处理分裂的事，还没来得及做其他事呢，不过现在可以了，在分裂之后，插入 Key (8) 就非常简单了。从根页面开始搜索，因为 8 比 20 小，所以还是从 Min 这个记录上找对应的叶子节点，找到的就是 101 号页面，然后在这个页面上做插入操作，这次已经满足插入的条件了，因为是排序的，所以插入到 5 和 10 的中间（这是为了可以简单直白地看到，其实内部的排序组织不是这样的），这样，101 号页面的数据就是 5、8、10 这三条记录了。

继续插入数据,接下来要插入的数据是 23,同样从根节点开始搜索,这个是大于 Key (20) 的,所以找到的相应叶子节点为 102 号页面,这个页面还有足够的空间,直接将 Key (23) 插入即可。继续插入 Key (22) 这条记录,同样的道理,插入完成之后,102 号页面的数据包括 20、22、23 三条记录。此时的 B+ 树结构及数据如图 8.5 所示。

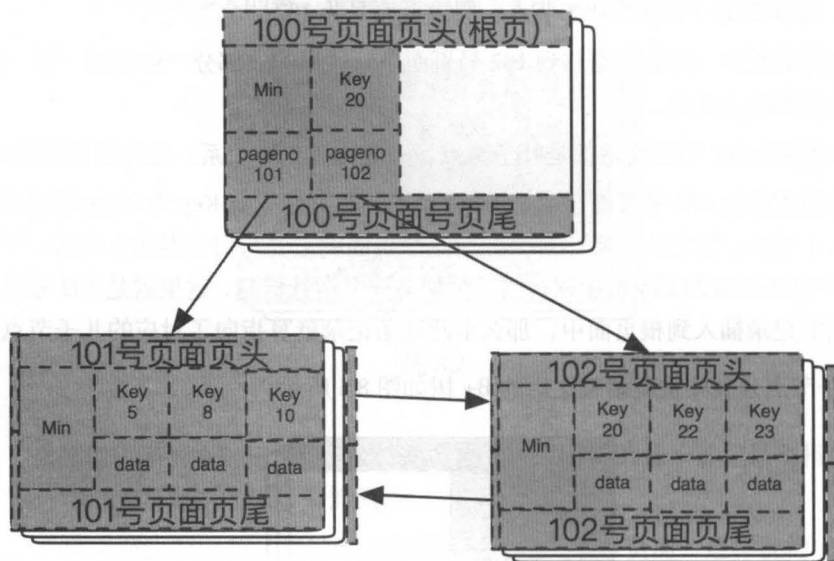


图 8.5

继续插入数据,此时插入的记录为 Key (50),因为它是大于 20 的,所以找到应该插入的叶子节点页面为 102。但此时发现 102 号页面已经满了,那就继续分裂吧,分裂属于叶子节点的分裂,上面已经做过了。同样地,先新建一个叶子节点页面,假设是 103 号页面,然后再移动一条记录 Key (23) 到 103 号页面,再将 103 号页面与根节点产生父子关系,也就是在根节点插入记录 (Key: 23, pageno: 103),这样根页面就产生了一个新的孩子节点 103。同时,103 号节点也就成了 102 节点的兄弟节点,需要用双向链表链起来,这样变换之后,新的 B+ 树结构如图 8.6 所示。

分裂之后,重新再次插入数据 Key (50),首先在根页面中搜索,发现他比 Key (23) 大,所以最终会插入到 103 号页面中,也就是 Key (23) 之后。

此时再插入下一个数据 Key (21),这就很简单了,直接插入到 102 号页面,插入之后,这个页面也就满了。

继续插入下一个数据 Key (53),同样的道理,插入到 103 号页面,插入之后,103 号页面的数据为 23、50、53,很明显,也已经满了。

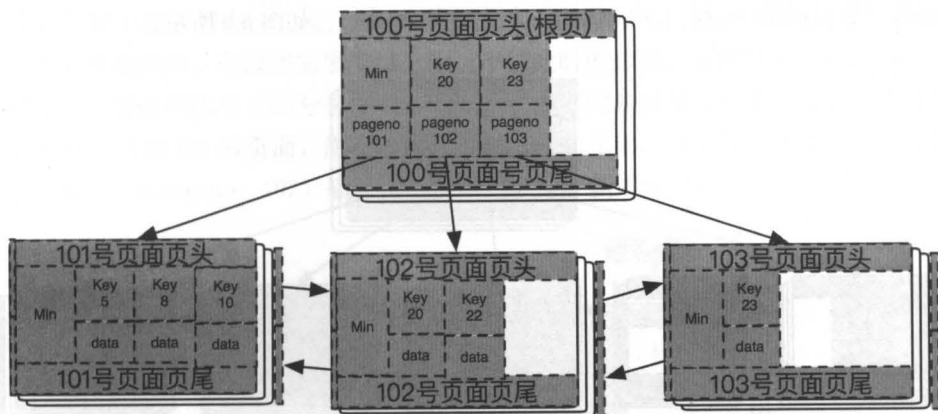


图 8.6

那么，继续插入下一个数据 Key (40)，同样因为它比 Key (23) 大，所以它需要插入到 103 号页面，但已经满了，所以此时再次发生叶子节点分裂的情况，这次就不细说了，假设这次申请的叶子节点为 104 号，分裂并插入之后的样子如图 8.7 所示。

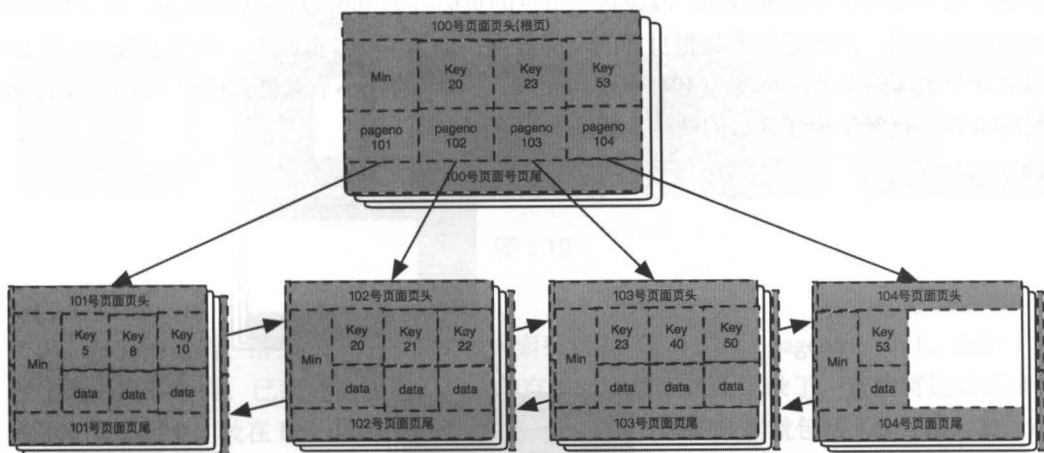


图 8.7

继续插入下一个数据 Key (9)，在根页面中，因为比 Key (20) 小，所以找到了 101 号插入目标页，但此时该页面已经满了，明显需要再次做叶子节点的分裂。同样地，创建一个新的叶子节点页面 105，移数据，使其成为兄弟节点，都做完之后，将 105 号页面中唯一一条 Key (10) (从 101 号页面中移过来的一条) 与根节点产生父子关系，创建一个索引记录 (Key: 10, pageno: 105)，并且插入到根页面中，不过此时发现有点惨，根页面已经满了，很明显可以想到，需要做根页面的分裂了。

我们先看一下目前的 B+ 树（不能称为一个完整的 B+ 树），如图 8.8 所示。

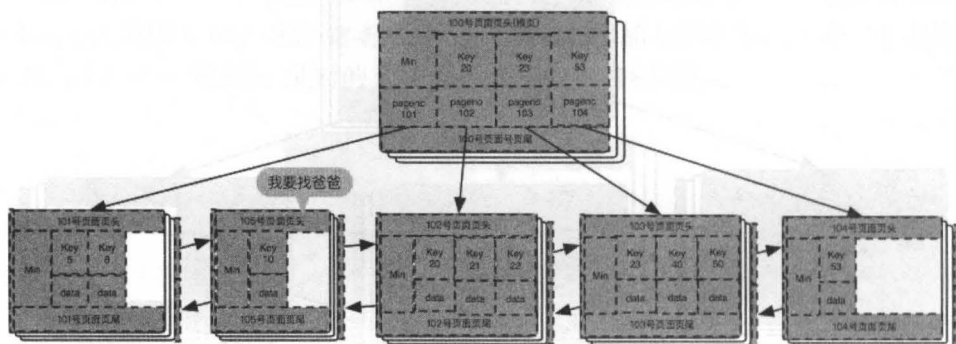


图 8.8

可以看到，105 号页面现在是没有父亲的，所以需要通过分裂来为它找到一个父亲。

此时要做的工作是将记录（Key: 10, pageNo: 105）插入到根节点，而根节点已经满了，所以需要再创建一个新的节点。新的节点应该创建在内节点页面上，因为要存储的数据是索引数据，而不是叶子节点的数据。假设这个节点的页面号为 106，这里需要注意，由于根页面始终是根页面，因此还是要把根页面的全部数据移动到 106 号页面中，相当于从 101 到 105 号这 5 个页面的父节点都变为 106 号了，而 100 号页面的最小节点也会指向 106 号页面。此时的 B 树，已经形成了 3 层的结构，样子如图 8.9 所示。

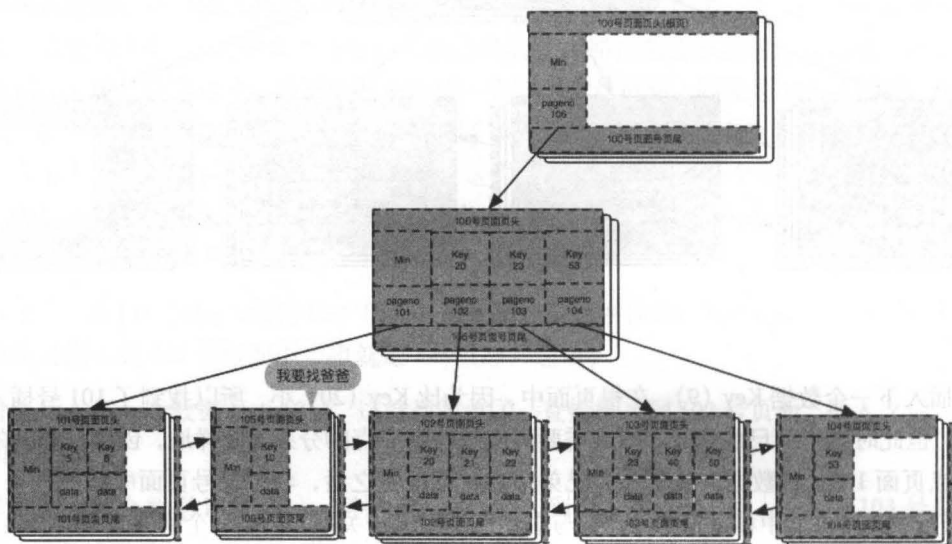


图 8.9

此时，继续做上面分裂之前的操作，也就是将数据（Key: 10, pageno: 105）写入 106 号页面中，但依然是满的，所以还需要继续分裂。不过此时的分裂，不叫叶子节点分裂，也不叫根节点分裂，而是叫内节点的分裂。内节点和叶子节点的分裂方法差不多，依然是创建一个新的节点，比如 107 号页面，然后作为 106 号页面的兄弟，移 1 条记录到 107 之后，将索引记录（Key: 53, pageno: 107）插入根页面中。这样，新的 B+ 树结构就是如图 8.10 所示这样的。

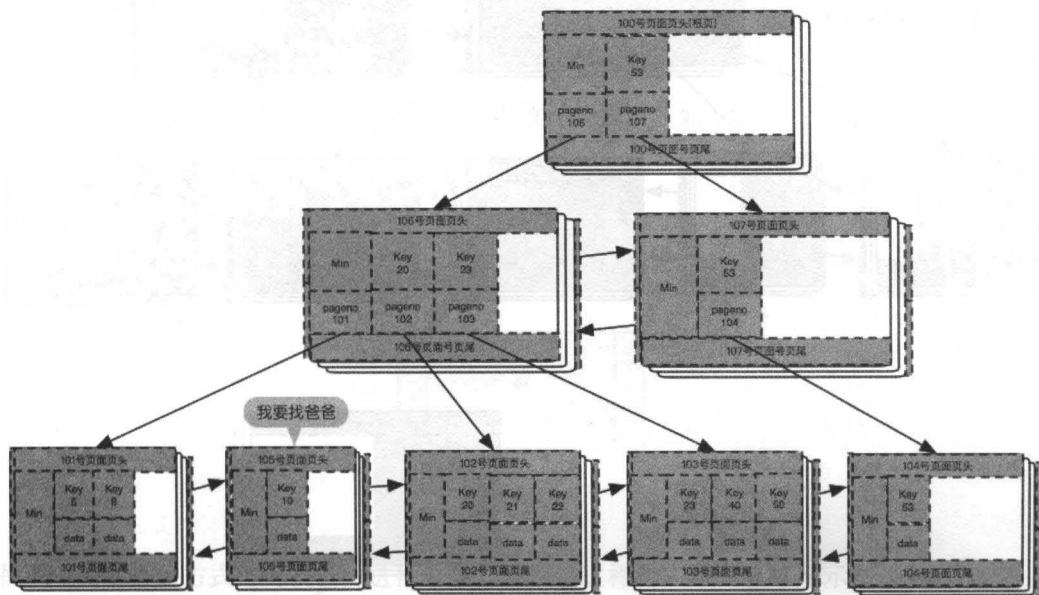


图 8.10

上面一步完成之后，继续在 106 号页面插入索引数据（Key: 10, pageno: 105）。此时，经过上面两次的分裂，已经成功找到了合适的空间可以放下这条记录了，直接将这条记录插入 106 号页面中，放在 Key (20) 之前，则此时 106 号页面的数据就包括 Key (10, 20, 23) 三条记录了，分别指向了 105、102、103 号页面，从而使 105 号节点也成功地找到了自己的“爸爸”。新的 B+ 树结构如图 8.11 所示。

关于 B+ 树的插入过程，就是这个样子，基本到了三层的话，就可以见到全部场景了，整个插入过程也都明确了。

不过别忘了，插入操作还没有完成，上面只是将索引指针记录插入到了内节点，两层节点分裂完成之后，又回到了插入 Key (9) 的操作上，因为这条记录还没有插入完成。此时很明显，还是会找到第 101 号页面，并且此时已经有足够的空间，直接插入即可，将所有数据插入完成之后的完整 B+ 树索引如图 8.12 所示。

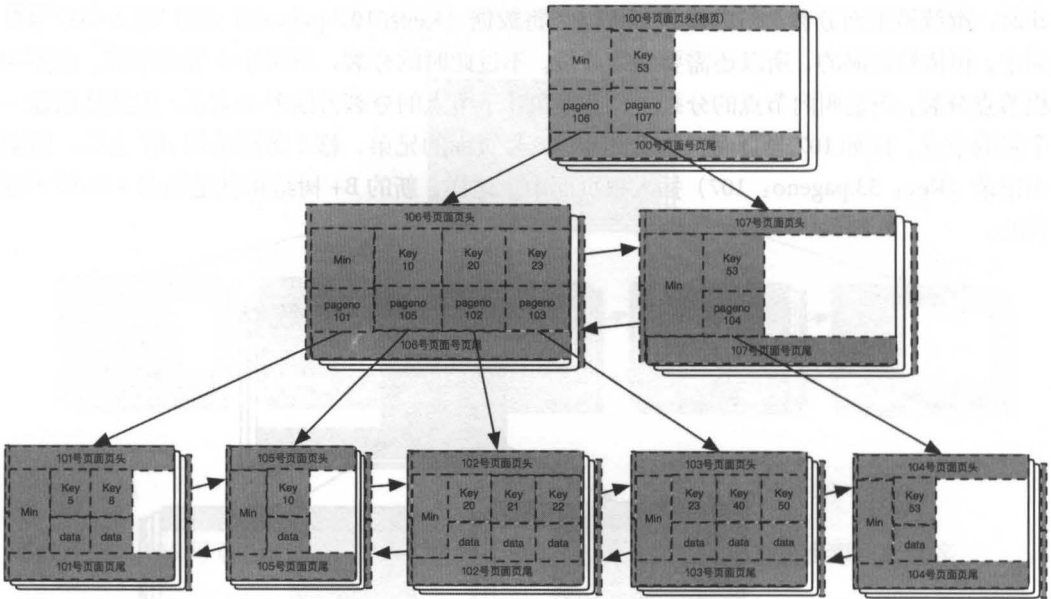


图 8.11

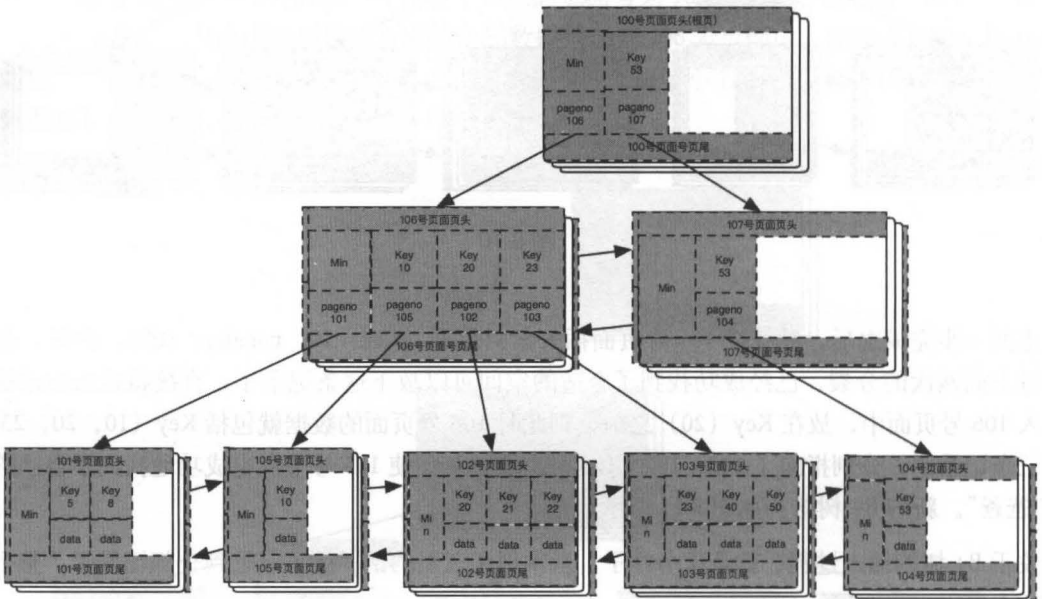


图 8.12

此时再回头看看，要插入的数据序列是：10、20、5、8、23、22、50、21、53、40、9，此时可以看一下这张图，从左到右把所有记录都读取出来，序列为：5、8、9、10、20、21、22、23、

40、50、53。很明显可以看到，相当于是给原始插入序列排了一次顺序，并且包含了全部数据，这也正是如果想要全表查询的话，在 InnoDB 内部，会直接定位到最左边的叶子节点上，然后依次从左到右将所有记录取出来的缘故，而这也正是我们所介绍的 B+ 树的特性。

一个页面至少要存储几条记录

有人问过我这样一个问题：为什么 InnoDB 数据页面中最少存储 2 条记录？一下子把我问住了，知道是这样，但从没有问过自己这是为什么。经过一番思考，知道了其中缘由，现在结合上面所讲的 B+ 树插入过程，分析一下这个问题的原因。

现在已经明白 B+ 树如何插入数据了，此时再做一些假设，如下。

- 在极端情况下，每一个页面中只能插入一条记录。
- 插入的数据就是 1、2、3。

现在来看一下一个简单的插入过程，如图 8.13 所示。

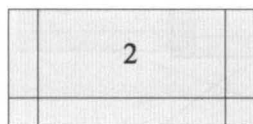


图 8.13

按照上面的分裂方式先对根页面进行分裂，结果是这样的，如图 8.14 所示。

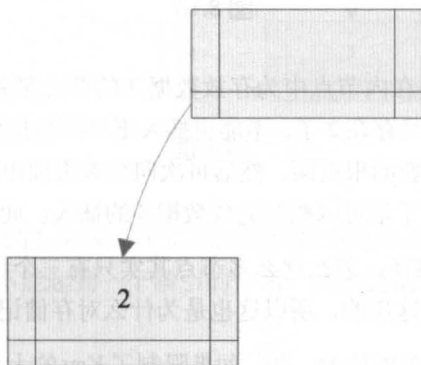


图 8.14

此时再向叶子节点插入数据 1，结果还是插入不进去。那么，接下来分裂叶子节点，创建新页面，新增一个 2 节点的右兄弟，而此时因为插入的是 1，比 2 小，所以需要将 2 的数据复制到新的叶子页面中，所以插入 1 之后的结果如图 8.15 所示。

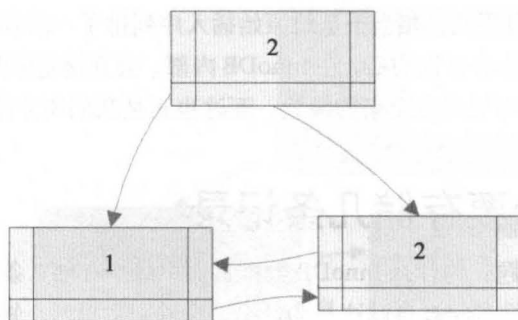


图 8.15

此时, 1 已经插入完成, 接下来插入 3。通过 B 树的搜索, 找到的插入位置在叶子节点 2 里面。但是因为只能插入一个记录, 所以这个节点需要再分裂, 因为插入的 3 比 2 大, 所以直接将 3 写入新叶子节点即可, 结果如图 8.16 所示。

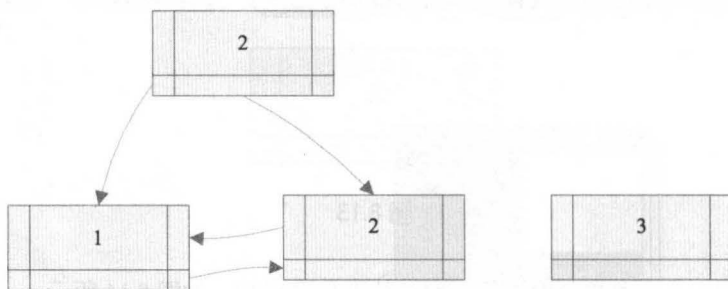


图 8.16

此时还是同样的道理, 需要在内节点中为存放数据 3 的节点创建索引键指针, 将 3 插入到根页面中, 但是根页面中已经存在 2 了, 不能再插入更多的数据记录, 因此根页面要再次分裂。同样的道理, 创建一个新的根页面, 然后再次向分裂页面中插入数据 3, 还是插入不成功, 那么再次分裂内节点, 于是可以顺利完成数据 3 的插入。此时的情况如图 8.17 所示。

图 8.17 是不是看上去觉得很怪, 怎么这么多节点其实只有三个记录? 是的, 如果一个页面只能插入一条记录, 就会是这样的, 所以这也是为什么对存储记录做了限制的原因。

但是还有一种情况, 现在将范围扩大一些, 如果限制了 Key 的大小, 一个内节点页面中至少要存储 2 条记录 (这里假设就是存储 2 条记录), 而叶子节点中只能存储一条记录, 那么此时的结果其实就是一个非常冗余的具有 B 树结构的双链表, 比如插入顺序为 1、2、3、4, 则结构如图 8.18 所示。

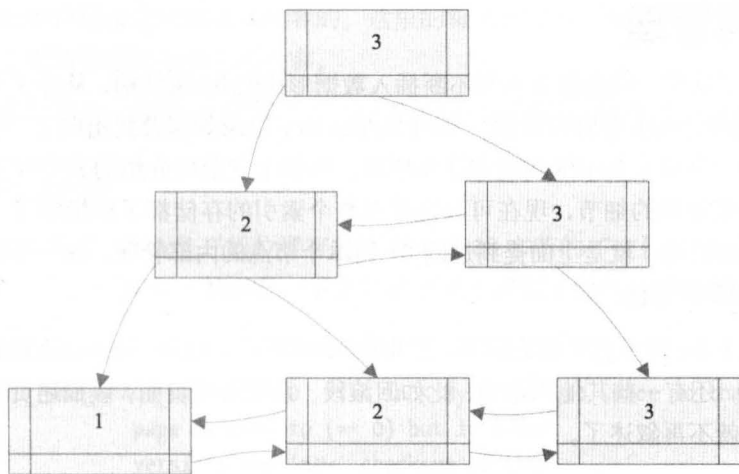


图 8.17

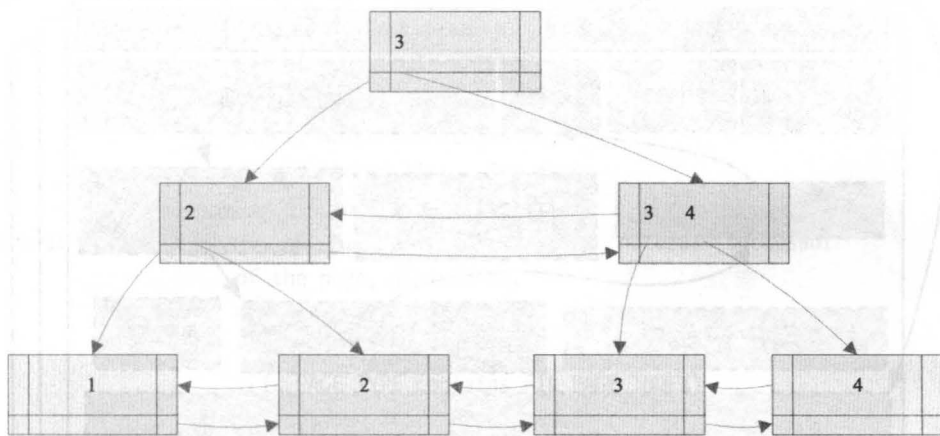


图 8.18

通过上面的实验，已经了解到，如果一个页面的数据量不能存储到 2 条记录，则这个 B+ 树就不能称为 B+ 树，因为它不能起到一个索引的作用，其实就是一个双向链表，但比双向链表占用的空间大很多。

页面中存储记录条数这个问题在数据库中是如上这样解释的，但归根到底，其实还是因为 B+ 树的特性所造成的，如果不存储至少 2 条记录，那么这个 B+ 树是没有意义的，形不成一个有效的索引。

页面结构管理

上面已经介绍了从单一的数据节点到不断插入数据形成的 B+ 树结构, 从叶子节点所存储的数据如何变成内节点中存储的数据, 从内节点的 Key 记录如何做到指向下一层 B+ 树节点等方面, 并一一介绍了其中的实现方法及原理。再加上之前所介绍的表空间文件管理方法中段、簇及页面管理的细节, 现在可以说是对整个索引的存储都了如指掌了。但还有一个知识点还没有介绍过, 就是上面提到过的 B+ 树单个节点的内部管理, 这一节就来介绍一下 InnoDB 的页面管理方法。

这里只讲述关于 B+ 树节点页面的细节, 因为在 InnoDB 中, 以 B+ 树节点这样的角色存在的页面是最多的, 还有一些其他用途的, 比如回滚段、系统管理页面、簇描述页等格式都是不相同的, 这些就不再叙述了。

先来看图 8.19。

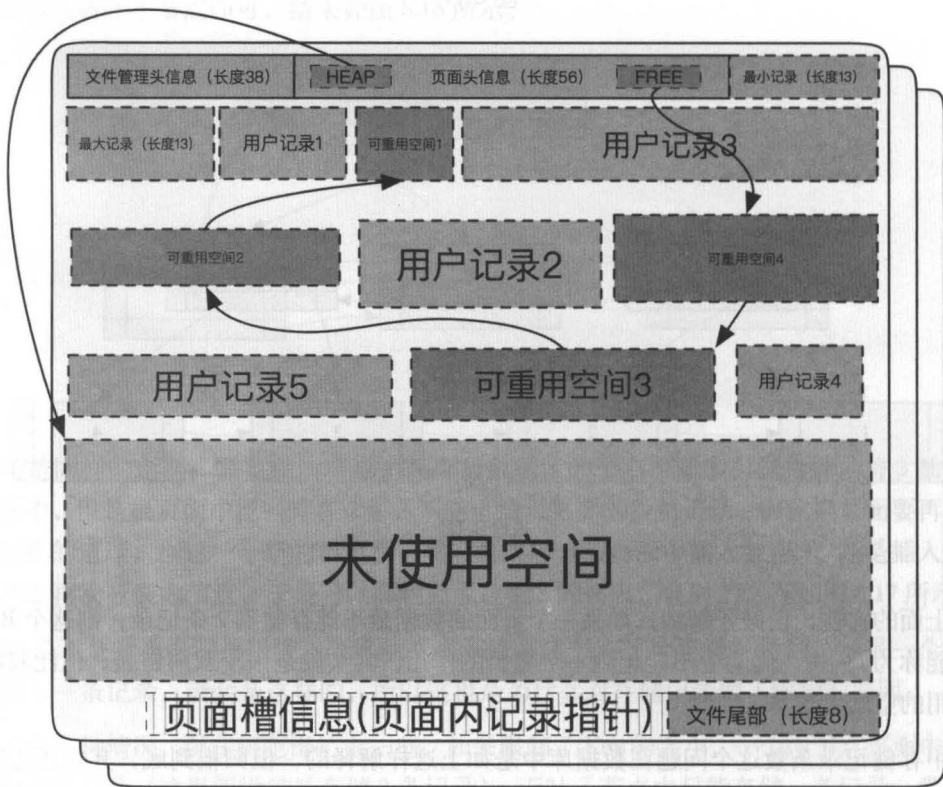


图 8.19

熟悉的同学, 应该一眼就可以看出来图 8.19 想要展示的内容是什么。没错, 就是 InnoDB 的页面内容。为了不引起误解, 也为了更形象地展示图中内容的关系, 制定了以下两条约定。

- 空闲空间 and 用户记录之间都是紧挨着的，这里的每条记录和可重用空间、空闲空间都分开，是为了更清楚地展示图中的内容。
- 将一个页面想象为一张纸，在存储完一行内容之后，可以换行，每条记录的长度大小不定。

文件管理头信息

从图 8.19 中可以看到，页面最开始位置存储了长度为 38 字节的“文件管理头信息”，这是每个页面都必须要有内容。下面看一下文件管理头中存储了哪些内容，先看一下在代码中定义的宏，如下。

```
#define FIL_PAGE_SPACE_OR_CHKSUM 0 /*!< in < MySQL-4.0.14 space id the
    page belongs to (== 0) but in later
    versions the 'new' checksum of the
    page */
#define FIL_PAGE_OFFSET 4 /*!< page offset inside space */
#define FIL_PAGE_PREV 8 /*!< if there is a 'natural'
    predecessor of the page, its
    offset. Otherwise FIL_NULL.
    This field is not set on BLOB
    pages, which are stored as a
    singly-linked list. See also
    FIL_PAGE_NEXT. */
#define FIL_PAGE_NEXT 12 /*!< if there is a 'natural' successor
    of the page, its offset.
    Otherwise FIL_NULL..
    B-tree index pages
    (FIL_PAGE_TYPE contains FIL_PAGE_INDEX)
    on the same PAGE_LEVEL are maintained
    as a doubly linked list via
    FIL_PAGE_PREV and FIL_PAGE_NEXT
    in the collation order of the
    smallest user record on each page. */
#define FIL_PAGE_LSN 16 /*!< lsn of the end of the newest
    modification log record to the page */
#define FIL_PAGE_TYPE 24 /*!< file page type: FIL_PAGE_INDEX,..., 2 bytes.
    The contents of this field can only
    be trusted in the following case:
    if the page is an uncompressed
    B-tree index page, then it is
    guaranteed that the value is
```

```

        FIL_PAGE_INDEX.
        The opposite does not hold.
        In tablespaces created by
        MySQL/InnoDB 5.1.7 or later, the
        contents of this field is valid
        for all uncompressed pages. */
#define FIL_PAGE_FILE_FLUSH_LSN 26 /*!< this is only defined for the
        first page in a system tablespace
        data file (ibdata*, not *.ibd):
        the file has been flushed to disk
        at least up to this lsn */
#define FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID 34 /*!< starting from 4.1.x this
        contains the space id of the page */
#define FIL_PAGE_SPACE_ID FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID
#define FIL_PAGE_DATA 38 /*!< start of the data on the page */

```

可以看到, 文件头信息最终到 FIL_PAGE_DATA 指示的位置便结束了。在这个位置之后所存储的信息, 分别做如下详细的解释。

- FIL_PAGE_SPACE_OR_CHKSUM: 长度为 4, 偏移为 0, 从注释中可以看到, 在 MySQL 4.0.14 版本之前, 这个位置是用来存储 Spaceid 的, 但都是为 0 的, 在之后的版本中, 存储了本页面的 CHECKSUM 值, 用来校验这个页面是不是完整, 是不是被损坏了。
- FIL_PAGE_OFFSET: 长度为 4, 偏移为 4, 表示的是这个页面在当前表空间中的页面号, 也就是以 16KB (默认) 为单位, 在页面内的偏移位置。页面号是从 0 开始计数的。
- FIL_PAGE_PREV: 长度为 4, 偏移为 8, 用来存储当前叶子节点的上一个页面, 如果已经是最左边的节点, 则这里存储为 FIL_NULL。
- FIL_PAGE_NEXT: 长度为 4, 偏移为 12, 用来存储当前叶子节点的下一个页面, 如果已经是最右边的节点, 则这里存储为 FIL_NULL。
- FIL_PAGE_LSN: 长度为 8, 偏移为 16, 用来存储当前页面最后一次被修改时, 对应日志的 LSN 值, 这个值在第 11 章会详细介绍, 与日志、页面刷盘、检查点等有密切关系。
- FIL_PAGE_TYPE: 长度为 2, 偏移为 24, 用来存储当前页面是什么类型的页面, 本节要介绍的类型为 FIL_PAGE_INDEX。
- FIL_PAGE_FILE_FLUSH_LSN: 长度为 8, 偏移为 26, 用来存储当前 InnoDB 存储引擎最大的被 FLUSH 到的 LSN 值, 在数据库正常关闭, 或者做检查点的时候, 都会将当前最新 FLUSH 的日志 LSN 写一次, 以保证在日志文件被删除, 或者修改大小的时候, 还可以找到一个正确的 LSN 值。从上面代码对应的注释中可以看到, 这个值只会被定义在每个表空间的第 0 号页面中, 也就是说只有第 0 号页面中的这个值是有效的, 其他所有页面的这个位置存储的数据才都是 0。

- `FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID`: 长度为4, 偏移为34, 在MySQL 4.1.x版本之后, 用来标识当前页面属于哪一个表空间。可以反过来想一下, 如果没有这个值, 那么当多个表同时缓存在 Buffer Pool 中, 并且在同一个页面号出现时, 就没办法区分它们对应的是哪个表空间了, 所以这个值是用来对应是哪一个表空间文件的。

页面头信息

从图 8.19 中可以看到, 在文件管理头信息之后, 就是页面头信息了, 这段信息总共占用 56 个字节。还是先来看一下代码, 这样会有一种踏实的感觉。



```
#define PAGE_HEADER FSEG_PAGE_DATA /* index page header starts at this offset */
/*-----*/
#define PAGE_N_DIR_SLOTS 0 /* number of slots in page directory */
#define PAGE_HEAP_TOP 2 /* pointer to record heap top */
#define PAGE_N_HEAP 4 /* number of records in the heap,
    bit 15=flag: new-style compact page format */
#define PAGE_FREE 6 /* pointer to start of page free record list */
#define PAGE_GARBAGE 8 /* number of bytes in deleted records */
#define PAGE_LAST_INSERT 10 /* pointer to the last inserted record, or
    NULL if this info has been reset by a delete,
    for example */
#define PAGE_DIRECTION 12 /* last insert direction: PAGE_LEFT, ... */
#define PAGE_N_DIRECTION 14 /* number of consecutive inserts to the same
    direction */
#define PAGE_N_RECS 16 /* number of user records on the page */
#define PAGE_MAX_TRX_ID 18 /* highest id of a trx which may have modified
    a record on the page; trx_id_t; defined only
    in secondary indexes and in the insert buffer
    tree */
#define PAGE_HEADER_PRIV_END 26 /* end of private data structure of the page
    header which are set in a page create */
/*----*/
#define PAGE_LEVEL 26 /* level of the node in an index tree; the
    leaf level is the level 0. This field should
    not be written to after page creation. */
#define PAGE_INDEX_ID 28 /* index id where the page belongs.
    This field should not be written to after
    page creation. */
#define PAGE_BTR_SEG_LEAF 36 /* file segment header for the leaf pages in
    a B-tree: defined only on the root page of a
    B-tree, but not in the root of an ibuf tree */
```

```
#define PAGE_BTR_IBUF_FREE_LIST PAGE_BTR_SEG_LEAF
#define PAGE_BTR_IBUF_FREE_LIST_NODE PAGE_BTR_SEG_LEAF
    /* in the place of PAGE_BTR_SEG_LEAF and _TOP
    there is a free list base node if the page is
    the root page of an ibuf tree, and at the same
    place is the free list node if the page is in
    a free list */
#define PAGE_BTR_SEG_TOP (36 + FSEG_HEADER_SIZE)
    /* file segment header for the non-leaf pages
    in a B-tree: defined only on the root page of
    a B-tree, but not in the root of an ibuf
    tree */
```

首先看一下每一行，PAGE_HEADER 表示的是页面头信息的开始位置，它的值等于 FSEG_PAGE_DATA 值，FSEG_PAGE_DATA 值又与 FIL_PAGE_DATA 值相同，FIL_PAGE_DATA 是上面已经见过的，其值为 38，所以这里也可以看到，页面头信息就是从偏移为 38 的位置开始的。

- PAGE_N_DIR_SLOTS: 长度为 2，偏移为 0，用来存储 Slot（槽）的个数，从图 8.19 中可以看到，在最下面有一部分就是槽的信息，具体是用来干什么的，后面会做详细介绍。
- PAGE_HEAP_TOP: 长度为 2，偏移为 2，用来存储当前页面中，还没有使用的空间的最小位置，也就是从这个位置开始到槽的位置，都是未使用的空间。这一点可以从图 8.19 中看到，在页面头信息中有一个指针出来，指向了未使用的空间。
- PAGE_N_HEAP: 长度为 2，偏移为 4，用来存储当前页面堆管理空间中存储的记录数目，这里面包括了最大记录和最小记录的管理。
- PAGE_FREE: 长度为 2，偏移为 6，用来存储当前页面中，已经被删除的记录所占用的空间组成的链表首指针。因为每一条被删除的记录，都有不定长度的空间，但这些空间还是可以被重新利用的，所以通过将一个页面中的所有这些空间组成一个链表，使它们被有效地管理起来，以便重新利用。这个首指针可以从图 8.19 中看到，并且链接了多个可重用空间。
- PAGE_GARBAGE: 长度为 2，偏移为 8，用来存储当前页面中，已经被标记为删除的记录数，这表示的是还没有真正被 PURGE 的记录数，如果有一条记录被 PURGE 了，这段空间就会被放到 PAGE_FREE 下面，同时 PAGE_GARBAGE 值也会相应地减去 1。
- PAGE_LAST_INSERT: 长度为 2，偏移为 10，用来存储当前页面中，最后被插入的记录的位置。
- PAGE_DIRECTION: 长度为 2，偏移为 12，与上面的 PAGE_LAST_INSERT 有关系，用来表示上次插入的方向。

- PAGE_N_DIRECTION: 长度为 2, 偏移为 14, 表示以同一个方向连续插入记录的次数。
- PAGE_N_RECS: 长度为 2, 偏移为 16, 用来存储当前页面中目前存储了多少条记录。
- PAGE_MAX_TRX_ID: 长度为 8, 偏移为 18, 用来存储在修改当前页面的所有事务中值最大的事务号。不过需要注意的是, 这仅仅在二级索引中才有意义。
- PAGE_LEVEL: 长度为 2, 偏移为 26, 用来存储当前节点在 B+ 树中处于第几层。根据之前所讲的, 就会明白这个值的意义了。不过在 InnoDB 中, 叶子节点的层数永远是 0, 从下层往上层计数。
- PAGE_INDEX_ID: 长度为 8, 偏移为 28, 用来存储当前页面属于哪个索引, 不过存储的是对应的索引 ID 值。
- PAGE_BTR_SEG_LEAF: 长度为 10, 偏移为 36, 用来存储 B+ 树叶子段的段头地址, 这个在之前关于表空间文件管理的内容中已经介绍过了。一个 B+ 树包括两个段, 叶子段和内节点段, 它们的首地址存储在每个 B+ 树的根页面中, 也就是这个位置。不过这里存储的是叶子段的地址, 下面的 PAGE_BTR_SEG_TOP 存储的是内节点段的地址。但是这两个信息只有在根页面中才有意义, 在其他页面中可以忽略不计。
- PAGE_BTR_SEG_TOP: 长度为 10, 偏移为 46, 存储内容参考前一个 PAGE_BTR_SEG_LEAF。

最小记录和最大记录

在 InnoDB 存储引擎中, 每一个 B+ 树索引页面都会有两个特殊的记录, 分别是“最大记录”和“最小记录”, 它们的作用是用来限定一个页面中, 数据记录的边界。“最小记录”是比任何一个记录都小的记录, 而“最大记录”是比任何一个记录都大的记录, 这两个信息有点类似页面头信息, 是固定位置并且固定大小的两条虚拟记录。

通过上面的讲述, 现在已经知道, 在索引内节点中, 每一层最左边页面上的最小记录, 都是用来承担索引作用的, 它指向的孩子节点中的数据, 在索引本身排序属性下, 比本节点中的数据都小。从图 8.19 中可以看到这一点, 这也是 B+ 树的特性。在 InnoDB 中, 使用了这个最小记录来承担这个责任。

另外, 还有一个作用就是在遍历页面数据时, 通过槽不断从前到后, 或者从后到前找数据的过程中, 只要找到了最大记录, 或者最小记录, 则说明这次搜索, 已经到了本页面的边界, 数据已经遍历完了, 这样就起到一个标记作用。

在图 8.19 中可以看到, 最大记录和最小记录都占用了 13 个字节, 不过这里有 5 个字节的浪费空间。实际上, 如果只是用来标记最大记录或最小记录的话, 则只需要 8 个字节即可, 但作为一个记录, 它还需要包括 5 个字节的记录头信息, 所以每条记录还需要加上这 5 个字节, 只不过对于它们而言, 这 5 个字节的记录头信息并没有什么用, 是浪费的。

页面数据空间管理

现在已经知道针对一个没有插入任何数据的 B+ 树页面来说, 已经存储的数据长度是 $38+56+13+13=120$ 字节。在这之后, 除了页面最后的 8 个字节用来存储页尾信息, 其余空间都是空闲的, 属于用户空间, 而此时 PAGE_HEAP_TOP 指向的位置就是偏移 $38+56+13+13=120$ 的位置。

在不断插入数据的过程中, 如果要插入这个页面, 则系统都会从这个页面的 HEAP 中申请需要的空间大小。比如目前是在偏移 120 的这个位置, 要插入的记录长度为 23, 则申请之后, 新的 PAGE_HEAP_TOP 指向的就是偏移 143 的位置。如果继续插入, 则继续申请, 继续更新 PAGE_HEAP_TOP 的值。可以看一下这块逻辑的代码, 如下。

```
UNIV_INTERN byte* page_mem_alloc_heap(
/*=====*/
    page_t*      page, /*!< in/out: index page */
    page_zip_des_t* page_zip,
    ulint*       heap_no) /*!< out: this contains the heap number
                           of the allocated record
                           if allocation succeeds */
{
    /* local variables... */

    /* 先获取本页面总共剩下的空间, 如果本页面的空间还有剩余, 则从本页面申请 */
    avl_space = page_get_max_insert_size(page, 1);
    if (avl_space >= need) {
        /* 从PAGE_HEAP_TOP位置开始申请空间 */
        block = page_header_get_ptr(page, PAGE_HEAP_TOP);
        /* 将新的位置更新到PAGE_HEAP_TOP */
        page_header_set_ptr(page, page_zip, PAGE_HEAP_TOP,
                           block + need);
        /* 将heap_no返回给上层 */
        *heap_no = page_dir_get_n_heap(page);
        /* 更新PAGE_N_HEAP的值 */
        page_dir_set_n_heap(page, page_zip, 1 + *heap_no);
        /* 返回申请出来的页面空间首地址, 但外面要使用的空间大小就是need */
        return(block);
    }
    return(NULL);
}
```

从代码中可以看到, 只要页面中有空间, 就会从堆 Heap 中申请, 这就是一个页面空间利用的逻辑。

这是空间的分配，那如果记录被删除并 PURGE 了，则系统会将这个记录对应的空间，通过 PAGE_FREE 来管理，每次在页面中删除记录之后，都会将新删除记录对应空间的 NEXT 指向原来 PAGE_FREE 指向的空间，然后再将 PAGE_FREE 指向新被删除的空间首地址，这样就通过这个链表管理起来了。因为在页面记录中，都会在记录的首地址前两个字节的位置存储当前记录的下一条记录，用来将记录之间做一个单向链表，那么自然而然被删除的空间也可以通过这个指针形成一个单链表，将所有的可重用空间串起来，最终通过 PAGE_FREE 管理起来。

关于记录之间的链表，如 8.20 图所示。

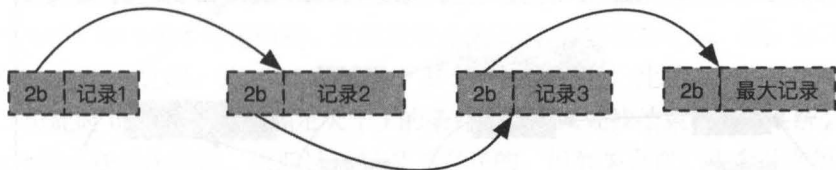


图 8.20

而此时可能会有同学要问，现在已经知道，未被分配的空间是通过 PAGE_HEAP_TOP 来管理的，而被删除并 PURGE 的记录是通过 PAGE_FREE 来管理的，那么被分配的记录是通过什么管理的呢？关于这一点，会在下一小节“经典的槽管理”中讲述。

经典的槽管理

讲清楚所有页面空间方面的问题之后，来讲一个最重要的问题，那就是在页面内部是如何来组织所有记录的。可能有人会问，如果从偏移 120 的位置开始申请插入一个记录 10，紧接着又插入一个 20，那如果再插入一个 15，此时页面内部的顺序是如何处理的，难道要将 20 向后平移吗？

类似这样的问题，在这一小节重会一一揭晓答案。

InnoDB 的槽与其他数据库，比如达梦的页面槽是不一样的，达梦的槽，是每一行记录对应一个槽，但 InnoDB 是多条记录对应一个槽，槽的作用是用来在页面内进行数据搜索的，因为查找一条数据时，是用 B+ 树来保证通过树形结构找到一个记录所在的页面，而在页面内部真正找到这条记录是通过槽来完成的。

槽本身是排序的，存储的位置从图 8.19 所示的页面图中可以看到，是在页面的最后位置，槽的长度与页面内存储的记录数有关系，一个槽占用两个字节。槽中数据的增长是以高字节到低字节的顺序存储的，最高位的槽代表的是页面内索引顺序最小的记录，而最低位的槽代表的是索引顺序最大的记录，也就是说，在页面内，是通过槽位置的顺序，来表示所有记录的顺序的，可以将槽理解为一个以下标值为元素值的可自由扩展的数组。那么很自然地，

这个数组是有顺序的，并且在不断增删改的过程中，都会修改这个数组，该平移的平移，该删除的删除，需要保证的是，每次操作完成之后，槽数组还是有序的。

上面说了，可以把槽理解为一个以下标为元素值的数组。当然，数组的真实元素值不会是下标，真正的值就是页面内槽所对应的记录在页面内的偏移量。所以，如果页面内数据发生了改变，只需要修改槽的位置即可变相地修改页面内数据的大小关系了。

此时再看一下页面管理的一张图，如图 8.21 所示。

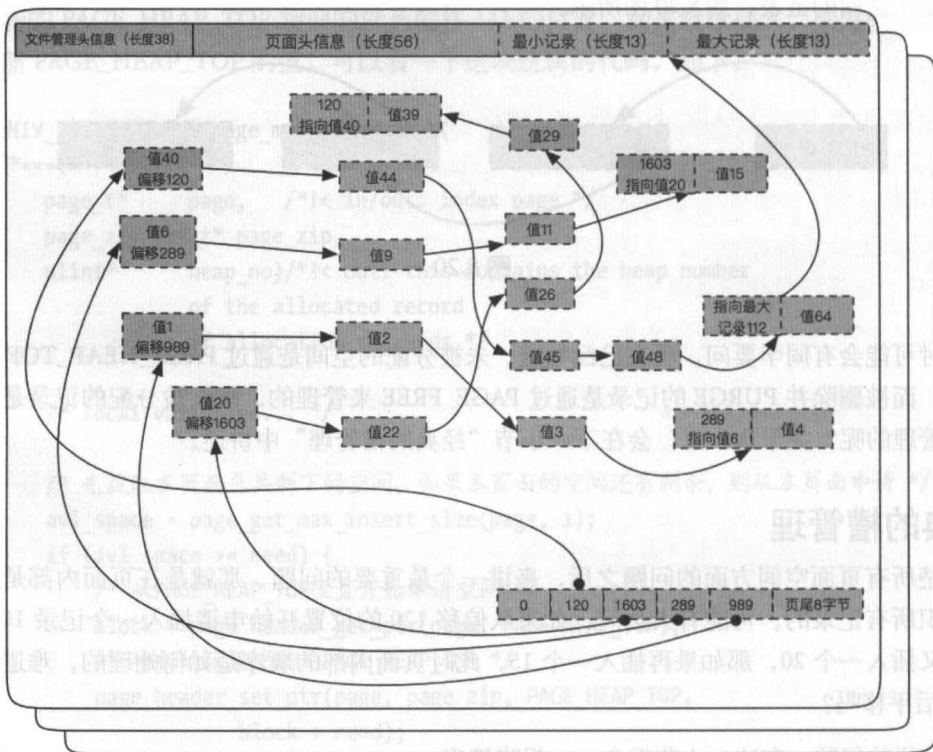


图 8.21

图 8.21 也是一个页面的示例图，主要侧重于槽与记录的关系。我们已经知道，在每条记录开始位置之前的两个字节，是用来存储下一条记录的指针信息的，所以在每一个槽指向的最后一条记录前面，为了体现它会指向下一个槽的第一条记录，都会有一个指向说明框，用来说明指向的值是多少、位置是多少。

在图 8.21 中可以看到，槽中存储的数据是没有顺序的，因为它们是槽所对应的第一个记录在页面内的偏移位置，比如第 0 号槽中存储的是 989，这说明本页面中最小的记录在偏移为 989 的位置存储，从页面中可以看到，这个槽用箭头指向的位置就是 989，对应的值是 1，其

他都比1大，所以这里是最小的，而这个槽对应的数据包括1、2、3、4四个值，通过链表连接起来，值为4的记录指向的下一个记录为偏移289位置的6，这个位置正好是第1号槽，那么值为4的记录也就是第0号槽的最后一记录。第1号槽包括的数据有6、9、11、15这四个值。依次类推，可以从槽数据出发，找到页面中的所有记录，并且数据都是排序的，只是槽本身存储的数据是无序的。

讲清楚这一点之后，假设现在要搜索25这个值，利用二分查找的办法，low（二分查找最低位标识）就是0，up就是槽的个数，当前位置的值是3（以0开始），二分值是1（ $0+3/2$ ），那就从第1号槽中去搜索，这个槽对应的数据是从6到15，显然不能满足。此时，向up的方向继续二分查找，此时的up还是3，low就变为上一次的二分值1了，二分之后，新的二分为2（ $1+3/2$ ），即从第2号槽中找，也就是位置为1603所指示的记录，我们发现这个槽对应的数据为22到39之间，此时22还是小于25，则继续二分，此时up为3，low为上一个二分值2，但此时up-low已经不满足大于1的条件了，所以查找结束，然后系统会从当前的low和up之间去找具体的值。很明显，25是不存在的，但如果存在，也会是在第2号槽中。

可以看一下实现这段逻辑的代码，在函数page_cur_search_with_match中，下面只看代码段。

```
/* Perform binary search. First the search is done through the page
directory, after that as a linear search in the list of records
owned by the upper limit directory slot. */
```

```
/* 先得到二分查找的两个边界值 */
low = 0;
up = page_dir_get_n_slots(page) - 1;
```

```
/* Perform binary search until the lower and upper limit directory
slots come to the distance 1 of each other */
```

```
/* 继续查找的条件是up - low > 1 */
while (up - low > 1) {
    /* 每次新获取二分值 */
    mid = (low + up) / 2;
    /* 取出二分值这个位置的槽所对应的记录 */
    slot = page_dir_get_nth_slot(page, mid);
    mid_rec = page_dir_slot_get_rec(slot);
```

```
/* 拿槽对应位置的记录与要查找的记录做对比 */
offsets = rec_get_offsets(mid_rec, index, offsets,
    dtuple_get_n_fields_cmp(tuple),
    &heap);
```

```

    cmp = cmp_dtuple_rec_with_match(tuple, mid_rec, offsets,
                                     &cur_matched_fields,
                                     &cur_matched_bytes);
    /* 如果要查找的值比中间值大，则下次查找的low就变为这次的二分值mid;
       如果要查找的值比中间值小，则下次查找的up就变为这次的二分值mid的 */
    if (UNIV_LIKELY(cmp > 0)) {
low_slot_match:
        low = mid;
        low_matched_fields = cur_matched_fields;
        low_matched_bytes = cur_matched_bytes;

    } else if (UNIV_EXPECT(cmp, -1)) {
up_slot_match:
        up = mid;
        up_matched_fields = cur_matched_fields;
        up_matched_bytes = cur_matched_bytes;

    } else if (mode == PAGE_CUR_G || mode == PAGE_CUR_LE) {
        goto low_slot_match;
    } else {
        goto up_slot_match;
    }
}

/* 循环结构之后，说明已经找到了一个大致的位置，up和low已经非常接近了。
   那么，此时就从low与up之间去遍历查找所有的记录，然后找到要查找的记录位置 */

/* 取出low号槽与up号槽，并且找到对应的记录值 */
slot = page_dir_get_nth_slot(page, low);
low_rec = page_dir_slot_get_rec(slot);
slot = page_dir_get_nth_slot(page, up);
up_rec = page_dir_slot_get_rec(slot);

/* Perform linear search until the upper and lower records come to
   distance 1 of each other. */

/* 在两个记录边界做循环查找记录的工作 */
while (page_rec_get_next_const(low_rec) != up_rec) {
    mid_rec = page_rec_get_next_const(low_rec);

```

```

offsets = rec_get_offsets(mid_rec, index, offsets,
                          dtuple_get_n_fields_cmp(tuple), &heap);

cmp = cmp_dtuple_rec_with_match(tuple, mid_rec, offsets,
                                &cur_matched_fields,
                                &cur_matched_bytes);
/* 如果要查找的记录比low大, 则继续向后查找;
   如果要查找的记录比low小, 则将上界缩小, 以减少循环次数 */
if (UNIV_LIKELY(cmp > 0)) {
low_rec_match:
    low_rec = mid_rec;
    low_matched_fields = cur_matched_fields;
    low_matched_bytes = cur_matched_bytes;

    } else if (UNIV_EXPECT(cmp, -1)) {
up_rec_match:
    up_rec = mid_rec;
    up_matched_fields = cur_matched_fields;
    up_matched_bytes = cur_matched_bytes;
    } else if (mode == PAGE_CUR_G || mode == PAGE_CUR_LE) {
        goto low_rec_match;
    } else {
        goto up_rec_match;
    }
}
/* 循环结束之后, 这里不管是找到还是找不到记录, 就都有一个结论了 */

```

从上面可以看到, InnoDB 代码也是使用了我们很熟悉的二分查找来实现了页面的数据搜索功能, 理论加实践才是硬道理。

页面尾部

除了文件尾部信息, 页面内其他信息都已经讲过了, 那么本小节就专门来讲述这最后的 8 个字节。

这 8 个字节承载的作用非常重要, 应该叫作“前后呼应”, 在页面最前面的位置, 也就是偏移为 FIL_PAGE_SPACE_OR_CHKSUM 的位置, 存储了页面校验的 CHECKSUM 值, 而这里为了验证页面的修改是不是完整。在最后 8 字节的前 4 个字节中, 存储了相同的内容, 用来与前面的值对比, 再加上读取页面时, 会根据页面内容重新计算一个 CHECKSUM 值出来, 将这三者做比较, 如果相同则说明这个页面是正确的, 否则说明有数据异常, 或者写入断裂等问题, 这就需要通过 InnoDB 的特性——两次写来保证了。在这 8 个字节的后面 4 个字节

中, 存储了当前页面最新被修改的 LSN 值, 也就是文件头信息中的 FIL_PAGE_LSN 值, 也是用来校验的, 意义与前者类似。

综上所述, 就是我们所熟知的 InnoDB Buffer 页的所有内容及管理方式, 知道这些之后, 应该会对日常的 MySQL 运维有所帮助。

页面重组

前面已经知道, 一个页面在管理的过程中, 都会涉及数据的不断插入和不断删除, 并且在插入数据时, 都会去已经删除的可重用链表中找合适的空间, 如果能放得下, 就会将新的记录放到这部分空间中。但久而久之, 如果写入删除频繁, 则很有可能会造成页面空间管理非常碎的情况, 所有的空闲空间大小加起来, 可以存储很多数据, 但是从 FREE 链表中, 找不到任何可以放得下一条记录的位置, 这种现象就叫作页面碎片。

页面碎片太多了, 一般会造成如下三个问题。

- 空间占用比较高, 但索引数据比较少。
- 在使用这个表的时候, 会导致大量的无效 IO, 因为很多空间都是碎片。
- 数据库整体性能变差。

一般可以通过下面的语句来查看一个表的空间使用率。

```
mysql> show table status like "%tablename%"\G
***** 1. row *****
      Name: tablename
      Engine: InnoDB
      Version: 10
      Row_format: Compact
        Rows: 6176
      Avg_row_length: 5945
      Data_length: 36716544
      Max_data_length: 0
      Index_length: 3162112
      Data_free: 11008999424
      Auto_increment: 4069237374
      Create_time: 2016-03-11 21:01:37
      Update_time: NULL
      Check_time: NULL
      Collation: utf8mb4_general_ci
      Checksum: NULL
      Create_options:
```

```
Comment: test table
1 row in set (0.00 sec)
```

从上面输出的信息可以很明显地看出来, Data_free 的大小是 11008999424, 而 Data_length 为 36716544。通过计算, 表碎片率达到 99.8%, 当然, 不能完全说这是碎片, 也有很多情况是一个大表在数据被删除后没及时回收空间造成的。

已经被删除数据的表本应该被及时处理回收空间, 否则占用的空间完全属于浪费。这种情况, 一般都会去做一个空的改表操作, 比如直接使用 `alter table tablename engine innodb`, 就可以将这个表的空间收回来, 相当于重组一次表。

而对于页面也是一样的道理, 在数据库向某一个页面插入时, 如果找不到大小合适的页面空间, 就会去做一次页面重组操作。重组的方式很简单, 就是新建一个 Buffer Pool 页面, 然后将碎片页面的数据一条一条地插入到新的页面, 因为复制记录的时候, 是从小到大进行的, 所以插入的数据也是一直往后追加的。插入完成之后, 将原来的 Buffer Pool 页面释放掉, 使新的内存空间对应上这个页面, 这样就做了一次重组。由于新页面中所有数据都是连续插入进去的, 这样使得空间完全没有浪费的情况, 最后一条记录后面的空间都在 PAGE_HEAP_TOP 下面管理着, 属于未使用空间, 如此便实现了页面的重组操作。

索引页面的回收

之前已经介绍过, 在数据插入时, B+ 树会不断地分裂, 从而导致这棵树会变成一个矮胖子, 那么数据如何被删除呢? B+ 树会做什么样的操作呢? 在这一节将会揭晓答案。

传统 B+ 树的数据删除, 一般都会有一个所谓的填充因子, 来控制页面数据的删除比例, 如果数据量小于这个填充因子所表示的数据量, 就会有节点合并, 这与分裂是相对应的, 实际操作起来比较复杂, 情况也比较多, 这方面的资料可以参考一些书籍来详细了解一下。在这一节中, 会专门针对 InnoDB 的实现, 来讲述页面的回收机制。

关于 InnoDB 记录删除, 分下面三种情况。

1. InnoDB 在删除数据时, 都会先检查当前页面剩余的记录数, 如果只剩下一条记录, 就直接将这个页面从 B+ 树中摘除, 也只有这种情况, InnoDB 才会回收一个页面, 这正是与上面所讲的传统 B+ 树的删除算法的不同所在。InnoDB 的页面根本不存在合并这么一说, 只有当页面剩下的数据只剩下 1 条, 并且这条数据正好又是要被删除的记录时, 这个页面才会被回收。当然, 这个页面不能是根页面, 根页面可以是空闲的, 即可以是空表。当然, 如果这一步中, 这个页面只剩下最后一条记录, 则可以直接将页面从 B+ 树中摘除, 下面两个也就不需要了。

2. 如果要删除记录所在的页面为内节点, 并且要删除的记录为本页面中的最左边记录, 也就是最小记录的下一条记录, 则说明此时的记录删除所影响的还有其父亲节点的指针。因为每个内节点的最左边记录, 都会在其父亲节点的某个位置存储着, 并指向这个要删除的记录, 所以此时需要更新其父亲节点的这条记录。
3. 做完上面两件事之后, 就可以放心地将本页面中要删除的记录删除。删除之后, 记录删除的操作就完成了。

接下来, 详细讲述一下上面步骤中的第一步, 看看是如何从一个 B+ 树中摘除一个页面的。

1. 首先判断当前页面的兄弟节点, 如果没有任何兄弟了, 就将本节点直接删除。但在这种情况下, 如果一层只有一个节点, 并且是空的, 则这棵 B+ 树就是空的, 这样就会递归地将整个 B+ 树释放掉, 最后只剩下一个空的根节点。
2. 根据本节点, 找到其父节点, 并且找到指向被删除节点的指针索引记录, 然后用上面所述的删除记录步骤, 将这条记录删除, 这样有可能会产生递归作用。
3. 删除上层索引记录之后, 将本节点从兄弟双向链表中摘除。
4. 将被删除的页面归还到本表空间的簇管理链表中去。

这是删除一个节点的步骤, 在上面删除记录时, 还有一个重要的问题没有讲述, 那就是第二步中删除最左边记录时导致的父亲节点中索引指针记录的更新。这也比较容易理解, 操作步骤如下。

1. 还是一样, 根据当前要删除的记录, 找到父亲节点中的索引指针记录, 然后通过一样的递归删除方法, 将这条记录删除。
2. 根据被删除记录的下一条记录, 构造一个新的索引指针记录, 指向的孩子节点不会改变。
3. 将上面构造的指针记录, 插入到当前节点的父亲节点中。这样就相当于更新了父亲节点中的索引指针记录, 从而维护了 B+ 树本身的特性。

上面所讲述的是在事务提交时, 将已经打了删除标志的记录做 PURGE 的时候, 才会从 B+ 树中真正删除记录的操作, 步骤列得比较简单, 但也说明了整体的 B+ 树退化的过程。

从上面的回收机制中可以看到, 相比传统的 B+ 树数据删除算法而言, 页面不做合并操作会相对简单一些, 容易操作。但也有缺点, 那就是删除数据时, 如果随机性比较大, 就会导致 B+ 树的空间浪费或碎片比较明显, 有些页面可能长期都会处于填充因子以下, 占用的空间比较大, 但实际存储的数据没有多少。所以, 对于 InnoDB 存储引擎的数据库表来说, 需要经常做一些巡检, 可以直接使用上面提到的语句 `alter table tablename engine innodb` 来整理, 空闲空间比较大的可以常做改表操作, 以便回收一些空间。

至此, B+ 树索引潮起又潮落, 算是讲述完毕了, 希望能帮助各位同学。

InnoDB 记录格式

背景

在第 8 章中，已经详细介绍了索引及页面格式，也简单介绍了关于索引记录内的一小部分细节，并且介绍了一条物理记录与一条索引记录在逻辑上是如何对应的。那么，在这一章中，就详细介绍一下关于物理记录的存储格式，以便更好地理解 InnoDB 在页面内对物理记录的管理方法。

关于页面格式，我们平常也会接触到，比如通过一条简单的命令，就可以看到创建的表使用的是什么类型的格式，如下。

```
mysql> show table status like "%my1%" \G
```

```
***** 1. row *****
```

```
      Name: my1
```

```
      Engine: InnoDB
```

```
      Version: 10
```

```
      Row_format: Compact
```

```
      Rows: 3
```

```
      Avg_row_length: 5461
```

```
      Data_length: 16384
```

```
      Max_data_length: 0
```

```
      Index_length: 16384
```

```
      Data_free: 0
```

```
Auto_increment: NULL
Create_time: 2016-12-05 12:24:16
Update_time: NULL
Check_time: NULL
Collation: utf8_general_ci
Checksum: NULL
Create_options:
Comment:
1 row in set (0.01 sec)
```

可以看到,第四列 Row_format 中对应的就是当前表使用的行格式存储类型。目前用得比较多的存储格式就是 Compact,这是 InnoDB 从 MySQL 5.0 版本中引入的,从名字可以看出,这是一种紧凑的、可以节省空间的格式。一般而言,对于 B+ 树方式的存储,一个页面存储越多的数据,一次 IO 就可以处理越多的记录,性能就会越高。所以,这里只介绍 Compact 类型的记录格式。

从源码入手了解行格式

在 MySQL 中,行格式有三种存储方式,如下。

- Server 层的格式,这种格式与存储引擎没有关系,即适用于所有存储引擎的格式,这种格式是操作数据时必然要经过的一种格式,也是 Row 模式下 Binlog 存储所使用的一种格式。
- 索引元组格式,这种格式是 InnoDB 存储引擎在存取记录时一种记录格式的中间状态,它要么是从 Server 层格式转换而来,要么是转换为 Server 层的格式。它是 InnoDB 在内存中存储的一种元组格式,也是内存中的一种用来存储所有列数据的数据结构,同一个表中,不同索引对应的元组是不同的,这种元组格式是与索引一一对应的。
- 物理存储记录格式,这种格式就是本节要重点介绍的一条记录在物理页面中的存储格式,即所谓的 Compact 格式。这种格式与上面的索引元组格式是一一对应的,每次存取,都是从 Server 层格式先转换为索引元组格式,然后再转换为在页面上的索引记录物理格式。而取数据也是一样,从索引页面中的物理格式转换为索引的元组格式,然后再转换为 Server 层格式,再做进一步处理。

可以通过一个最普遍的插入操作来跟踪 InnoDB 的记录格式,因为在插入时,系统接收到的是公共的 MySQL 记录格式 record (即 Server 层格式),它没有涉及任何存储引擎,也就是说不管当前这个表对应的存储引擎是什么,记录格式是一样的。对于插入,MySQL 函数对应的是 ha_write_row,具体到 InnoDB 存储引擎,实际调用的函数是 ha_innobase::write_row。在这里,InnoDB 首先会将接收到的 record 记录转换为自己的一个元组 tuple (索引元组格式),

这其实是与 record 对应的 InnoDB 的表示方式，它是一个内存的、逻辑的记录，在系统将其真正地写入到页面之前，这条记录的存在方式都是这个 tuple。

下面主要从源码的角度来研究 InnoDB 如何将一个 tuple 转换为它的物理存储记录，主要研究代码的实现逻辑及记录的格式。

在某一个页面插入一个元组（一条记录），实现这个操作的函数是 `page_cur_tuple_insert`，它的参数就是一个 `dtuple_t*` 类型的 tuple。在这里，它首先要分配一片空间来存储将要转换过来的物理记录，所以，这里需要先计算空间的大小，计算方法如下。

- 首先，每条记录都包括下面两部分：`REC_N_NEW_EXTRA_BYTES + UT_BITS_IN_BYTES (n_null)`，前面表示的是这种格式固定长度的 extra 部分，这部分用来存储的内容在后面会给出；后面表示的是所有字段中哪些字段的值是 null，当然这里只存储那些 nullable 属性的字段，如果创建表的时候指定为 not null 的话，就不会被存储，此处是用一个位来表示一个字段的 null 属性。这部分被系统代码命名为 `extra_size` 变量值。
- 统计每一个列中数据的长度，在统计这个信息的时候，又有多种情况，主要分定长字段和变长字段。对于定长字段而言，它的长度直接就是数据类型的长度，比如 int 类型的就是 4 个字节、rowid 列就是 6 个字节等，没有其他附加长度；对于变长字段而言，除了数据内容本身的长度外，还需要计算其数据长度的存储空间。且如果变长字段的字义长度大于 255 个字节，或者字段的数据类型为 BLOB，就需要用 2 个字节来存储这个字段的长度；如果定义长度小于 128 个字节，或者小于 256 个字节且类型不是 BLOB 类型，那么这个字段的数据长度用一个字节来存储，除了上面 2 种情况之外，都用 2 个字节来存储。在这一部分中，用来存储变长字段数据长度的空间大小也被 InnoDB 计算到 `extra_size` 中。

所以现在可以知道，一个 InnoDB 的记录包括两部分，一部分是 `extra_size`，另一部分是数据内容，这 2 部分的总长度就是上面计算出来的结果，这里把它定义为 `record_size`。

接下来，申请空间，进行元组到物理存储记录的转换工作。

转换函数为 `rec_convert_dtuple_to_rec_new`，参数有申请好的记录空间 `buf`、元组和索引的内存结构。源码如下。

```
/* Builds a new-style physical record out of a data tuple and
   stores it beginning from the start of the given buffer.
   @return pointer to the origin of physical record */
static rec_t* rec_convert_dtuple_to_rec_new(
/*=====*/
    byte*      buf,      /*!< in: start address of the physical record */
    const dict_index_t* index, /*!< in: record descriptor */
```

```

const dtuple_t*      dtuple) /*!< in: data tuple */
{
    ulint    extra_size;
    ulint    status;
    rec_t*   rec;

    status = dtuple_get_info_bits(dtuple) & REC_NEW_STATUS_MASK;
    /* 如上所述, 这是用来计算记录头大小的, 记录头大小用extra_size来表示 */
    rec_get_converted_size_comp(
        index, status, dtuple->fields, dtuple->n_fields, &extra_size);
    /* 因为buf是用来存储整个记录的开始位置的, 这里的buf + extra_size
       表示的就是存储第一个列的位置了, 即rec所指的位置 */
    rec = buf + extra_size;

    /* 将元组dtuple转换为页面上的Compact格式记录 */
    rec_convert_dtuple_to_rec_comp(
        rec, index, dtuple->fields, dtuple->n_fields, status, false);

    /* Set the info bits of the record */
    rec_set_info_and_status_bits(rec, dtuple_get_info_bits(dtuple));

    return(rec);
}

```

在上面的代码中, 可以看到有一行操作是 `rec = buf + extra_size`, 变量 `rec` 表示的是数据内容 (列数据) 存储的开始位置, `extra_size` 就是上面计算方法中所说的 `extra_size`。

那么, 真正执行转换的是接下来调用的 `rec_convert_dtuple_to_rec_comp` 函数, 下面是精简之后的代码。

```

/* Builds a ROW_FORMAT=COMPACT record out of a data tuple. */
void rec_convert_dtuple_to_rec_comp(
    /*=====*/
    rec_t*      rec,      /*!< in: origin of record */
    const dict_index_t* index, /*!< in: record descriptor */
    const dfield_t* fields, /*!< in: array of data fields */
    ulint        n_fields, /*!< in: number of data fields */
    ulint        status /*!< in: status bits of the record */
)
{
    /* local variables */

```

```

/* 上面的rec所指的位置,就是上面所说的存储了第一列数据的位置,在
   这个位置前面,存储的就是extra_size包含的所有数据 */
/* 下面这个计算,用来找到记录头中,用于存储nulls标志位的位置。
   这里是在rec所指的位置上向低位位移6个字节开始写入,并且可以看到,下面
   写入的顺序是从高位向低位写入(都是nulls--),所以可以知道nulls的存储
   与列的顺序是相反的,也就是nulls标志位的信息,与列数据存储之间,相差
   REC_N_NEW_EXTRA_BYTES个字节,也就是5个字节 */
nulls = rec - (REC_N_NEW_EXTRA_BYTES + 1);
end = rec;
n_null = index->n_nullable;
/* 与上面类似的是,通过nulls标志信息的位置,计算出来存储变长长度
   信息的位置。因为在一个索引中,有多少个nullable的列是固定的,所以
   可以直接计算出来。那么,此时从nulls标志位开始再向低位位移固定字节数,
   即可找到长度存储的位置,变长列长度的存储顺序与列的顺序也是相反的 */
lens = nulls - UT_BITS_IN_BYTES(n_null);
/* clear the SQL-null flags
   lens + 1为nulls标志存储的开始位置, nulls - lens为nulls存储的长度*/
memset(lens + 1, 0, nulls - lens);
/* 因为从函数rec_convert_dtuple_to_rec_new中可以看到,此处的rec是
   用来存储第一个列数据的位置,这个位置是从记录的首地址向高位移动了
   extra_size个字节的位置,所以此时要通过rec这个位置来向低位移动,来找到
   nulls及lens的存储位置,是预先算好的,还在extra_size所管理的范围内*/

/* Store the data and the offsets
   将每一个列的数据,以及NULL和LEN的信息存储到相应的位置 */
for (i = 0, field = fields; i < n_fields; i++, field++) {
    const dict_field_t* ifield;
    type = dfield_get_type(field);
    len = dfield_get_len(field);
    /* 如果是一个索引指针,则直接使用4个字节来存储,即
       REC_NODE_PTR_SIZE所表示的长度,也算是固定长度 */
    if (UNIV_UNLIKELY(i == n_node_ptr_field)) {
        memcpy(end, dfield_get_data(field), len);
        end += REC_NODE_PTR_SIZE;
        break;
    }

    /* 计算NULL信息,因为这个标志是通过位来存储的,所以
       对每一个字节都需要做位处理 */
    if (!(dtype_get_prtype(type) & DATA_NOT_NULL)) {
        /* nullable field */

```

```

        if (UNIV_UNLIKELY(!(byte) null_mask)) {
            nulls--;
            null_mask = 1;
        }

        /* set the null flag if necessary */
        if (dfield_is_null(field)) {
            *nulls |= null_mask;
            null_mask <= 1;
            continue;
        }

        null_mask <= 1;
    }
    ifield = dict_index_get_nth_field(index, i);
    fixed_len = ifield->fixed_len;
    if (temp && fixed_len
        && !dict_col_get_fixed_size(ifield->col, temp)) {
        fixed_len = 0;
    }
    /* 这里, 即为大家经常讨论的, 实现列的大小和存储其长度字节数动态匹配的位置。
       根据这段代码可以来看清楚其逻辑, 下面这段注释也可以看出一二 */
    /* If the maximum length of a variable-length field
       is up to 255 bytes, the actual length is always stored
       in one byte. If the maximum length is more than 255
       bytes, the actual length is stored in one byte for
       0..127. The length will be encoded in two bytes when
       it is 128 or more, or when the field is stored externally. */
    if (fixed_len) {
        /* 如果是定长数据, 就直接不需要存储其长度了 */
    } else if (dfield_is_ext(field)) {
        /* 如果是行外存储数据, 需要两个字节来存储其长度 */
        *lens-- = (byte) (len >> 8) | 0xc0;
        *lens-- = (byte) len;
    } else {
        /* 如果变长列定义长度小于128, 则使用1个字节来存储其长度 */
        /* 如果变长列定义长度小于256, 且其类型不是BLOB,
           则使用1个字节来存储其长度 */
        if (len < 128
            || (dtype_get_len(type) < 256
                && dtype_get_mtype(type) != DATA_BLOB)) {

```

```

        *lens-- = (byte) len;
    } else {
        /* 其他情况下, 都使用2个字节来存储其长度 */
        *lens-- = (byte) (len >> 8) | 0x80;
        *lens-- = (byte) len;
    }
}

/* 将元组列信息, 写入到Compact记录的对应列中去, len为其对应的存储长度 */
memcpy(end, dfield_get_data(field), len);
end += len;
}
}

```

从上面的代码中, 可以大概了解到一个元组是如何一点点向 Compact 记录格式转换的, 并且大致知道了这种 Compact 格式的样子, 不过还是再来重点关注一下下面这段代码。

```

if (!(dtype_get_prtype(type) & DATA_NOT_NULL)) {
    /* nullable field */
    if (UNIV_UNLIKELY(!(byte) null_mask)) {
        nulls--;
        null_mask = 1;
    }
    if (dfield_is_null(field)) {
        *nulls |= null_mask;
        null_mask <<= 1;
        continue;
    }
    null_mask <<= 1;
}
}

```

第1行的条件 `!(dtype_get_prtype(type) & DATA_NOT_NULL)`, 判断的是这个列在建表时指定的 NULL 属性, 如果为 True, 则说明是 nullable 的。满足条件之后, 处理的就是记录中的 nulls 信息, 从这里也可以看到, 记录头中只存储了 nullable 列的信息, 而 not null 是没有存储的, 这也是 NULL 和 NOT NULL 在存储方面上的区别之一。

第3行表示的是当 `((byte)null_mask)` 为 0 时, nulls 向前退一个字节, 并且将 null_mask 恢复为 1 的初值。因为一个字节有 8 位, 如果 `(byte)null_mask` 为 0, 则说明 `null_mask <<= 1` 移了 8 次, 高位是 1, 低 8 位都是 0, 虽然是 int 类型的, 但转换为 byte 类型, 其值就是 0, 也就是已经处理了 8 个 nullable 的列, 此时就会用 extra_size 中的下一个字节来存储 nulls 标志了。

第7行表示的是如果这个列的数据就是 null 值, 那就需要将这个 NULL 值信息反映到 nulls 数组中去。null_mask 的值与当前 nulls 字节是时刻对应的, null_mask 的二进制值中, 只有

一个位是 1，如果某个列的值为 NULL，则当前列在所有的 nullable 列中的相对位置就会与这个 1 所对应，此时就可以通过 `*nulls|= null_mask` 将这个列的 NULL 标志写入到 nulls 标志信息中去，这样就表示了哪一列的值是 NULL 了。

所以从这里可以看出，整个 nulls 空间中的位图是以从后向前的顺序来表示所有 nullable 列的 null 信息的。

并且此时已经可以详细地知道，所谓的 Compact 究竟是什么样子的了，如图 9.1 所示。

```
|-----extra_size-----|-----fields_data-----|
|--columns_lens--|--null_lens--|--fixed_extrasize--|--col1---|--col2---|--col3---
```

图 9.1

从图 9.1 中可以看到，前一部分 extra_size 包括三部分，第一部分是 columns_lens，即上面所提到的用来存储变长列数据长度的空间；第二部分就是用来存储所有 nullable 列中存储了 NULL 值的列空间；还有一部分，就是上面代码中所见过的长度为 5 (REC_N_NEW_EXTRA_BYTES) 的一段空间。

在 extra_size 之后，就是每一列数据真正占用的空间了，每一列在记录中的物理存储顺序，就是所定义表的列的顺序，一列一列一直向后追加。

至于剩下还未介绍的 5 个字节用来做什么，可以通过下面的表格来了解。

名称	大小 (bit)	说明
预留位 1	1	没有使用
预留位 2	1	没有使用
delete_mask	1	用来标记当前记录是不是已经被打上了删除标记
min_rec_mask	1	最小记录的标志
n_owned	4	用来表示当前槽所管理的记录数，槽的解释参见第 8 章相关介绍
heap_no	13	当前记录对应的 heap_no 号，在页面堆中的位置信息
record_type	3	用来表示当前记录的类型，0 表示的是普通记录，1 表示的是 B+ 树内节点指针记录，2 表示的是最小记录，3 表示的是最大记录
next_record	16	用来将页面中的所有记录都连接起来，形成一个单链表，在第 8 章中也介绍过，重用记录空间也是通过这个指针来组织的，一个槽对应的多条记录，也是通过这个指针来组织的
总共	40	这就是上面介绍的 5 个字节中，每一位的用途

到这里，一条记录从逻辑到物理的转换就完成了，从中也知道了 InnoDB 是如何实现其物理记录的存储的。

总结

InnoDB 的代码非常优美，但也非常精练，所以有些地方很难一下子看懂，需要揣测、体会才能深入理解。同时，很多地方是直接硬编码的，这样导致理解更加困难，最好的方式是通过宏将其命名，有助于理解。

10

揭秘独特的两次写

两次写 (InnoDB Double Write) 是 InnoDB 中很独特的一个功能点。本章计划从源码的角度, 详细阐述两次写的原理和实现方法。

首先, 说明一下为什么会有两次写这个东西。

因为 InnoDB 中的日志是逻辑的, 所谓逻辑就是比如插入一条记录时, 它可能会在某一个页面 (这条记录最终被插入的位置) 的多个偏移位置写入某个长度的值, 例如页头的记录数、槽数、页尾槽数据、页中的记录值等。这些本是一些物理操作, 而 InnoDB 为了节约日志量及其他一些原因, 设计为逻辑处理的方式, 即在一个页面上插入一条记录时, 对应的日志内容包括表空间号、页面号、将被记录的各个列的值等内容, 在真正物理插入的时候, 才会将日志逻辑操作转换为前面的物理操作。

但这里的一个问题是, 如果那个页面本身是错误的, 这种错误有可能是因为写断裂 (1 个页面为 16KB, 分多次写入, 后面的有可能没有写成功, 导致这个页面不完整) 或者其他原因引起的, 那么这个逻辑操作就没办法完成了, 因为完成此操作的前提是这个页面还是正确的、完整的。如果这个页面不正确的话, 里面的数据是无效的, 就可能会产生各种不可预料的问题。

因此首先必须要保证这个页面是正确的, 方法就是两次写, 它的实现思想是一种备份思想, 更直白地讲, 是一种镜像。

下面就它的实现方式说明一下, 关于两次写这个特性, 在 MySQL 5.5 到 MySQL 5.7 的升级过程中, 变化是比较大的, 这里只讲述最新版 5.7 的实现方式。

两次写，包括两部分，一部分是对单独一个页面刷盘时的两次写，另一部分是批量刷盘时的两次写，下面就这两种刷盘类型分别介绍对应的两次写实现机制。

单一页面刷盘

单一页面刷盘，实际上是 MySQL 5.5 版本中的实现方式。MySQL 会在系统页面，也就是在 ibdata 文件的第五个页面，同时还是用来存储事务信息的一个页面中存储关于两次写的信息，偏移位置是页面结束位置的最后 200 个字节处，存储的信息主要有下面的内容。

```

/*这里存储的是两次写页面所在段的地址信息 */
#define TRX_SYS_DOUBLEWRITE_FSEG      0
/*!< 用来判断是不是已经初始化过两次写页面 */
#define TRX_SYS_DOUBLEWRITE_MAGIC      FSEG_HEADER_SIZE
/*!两次写页面第一个簇的首地址，两次写页面总共有2个簇，一个簇为64个页面*/
#define TRX_SYS_DOUBLEWRITE_BLOCK1    (4 + FSEG_HEADER_SIZE)
/*!第二个簇的首地址*/
#define TRX_SYS_DOUBLEWRITE_BLOCK2    (8 + FSEG_HEADER_SIZE)
/*!< 将上面的MAGIC、BLOCK1与BLOCK2重复存储，防止页面自己的不完整*/
#define TRX_SYS_DOUBLEWRITE_REPEAT    12

```

在第一次建库的时候，会分配一个段的空間，段的地址会存储到两次写信息偏移 TRX_SYS_DOUBLEWRITE_FSEG 的位置，每次使用两次写机制写数据时，都会从这个位置读取到段的位置，进而找到段的首地址。

在偏移 TRX_SYS_DOUBLEWRITE_MAGIC 的位置，存储的是用来验证当前两次写是不是正常或是不是已经申请的标志，这一点可以不用太关注。

偏移 TRX_SYS_DOUBLEWRITE_BLOCK1 和 TRX_SYS_DOUBLEWRITE_BLOCK2 存储的是两次写空间的位置，它们在 ibdata 文件中属于同一个段，在初始化数据库时会确定具体位置，是用来真正存储两次写页面数据的空间，它们对应的空间大小都是一个簇（被同一个段管理），占用磁盘空间分别为 1MB。一个簇包括 64 个页面，每个页面为 16KB， $16KB \times 64 = 1MB$ ，两次写总共包括 2MB（默认值）的数据，这便是两次写中的 2MB 数据的由来。

那么，初始化所要做的工作就是将上面的信息补充完整，BLOCK1 与 BLOCK2 分别对应 2 个簇的首地址，同时还要申请两个簇对应的内存空间来缓存两次写数据。

偏移 TRX_SYS_DOUBLEWRITE_REPEAT 所指的位置用来将前面三个信息：MAGIC、BLOCK1、BLOCK2 重复存储一次，这样可以保证数据的安全性，降低页面本身出现数据错误的可能性。这一点不用太关注。

除了上面的信息需要持久化到文件中之外，还会有一个空间用来存储这 128 个页面的页面信息，这是在内存中的，每次刷盘前，都会将要刷盘的页面信息临时保存到数组中，有了这

些信息,单个页面刷盘的两次写就已经足够且可以正常运转了,这个缓存,可以被称为两次写缓存数组。

下面说明一下它的使用过程。在做页面刷盘的时候,如果开启了两次写的功能,InnoDB 要做的就不是简单地直接将数据做 IO 操作写入到硬盘,而是先在两次写缓存数组中,找到一个空闲位置,并将这个位置标记为已使用状态,然后再把整个页面的数据复制到空闲位置对应的缓存空间中。复制完成之后,系统会将这个页面的数据刷到两次写文件中,也就是 `ibdata` 文件中。当然,复制的数据量是一个页面的大小,偏移位置是这个页面在两次写缓存空间中的位置,对应着 `TRX_SYS_DOUBLEWRITE_BLOCK1` 或 `TRX_SYS_DOUBLEWRITE_BLOCK2` 中的位置。因为两次写缓存数组是 128 个元素,而对应的 `TRX_SYS_DOUBLEWRITE_BLOCK1` 及 `TRX_SYS_DOUBLEWRITE_BLOCK2` 也是 128 个页面,它们是一一对应的,所以具体刷什么位置,可以很容易地计算出来。

在将单一页面刷到 `ibdata` 文件的两次写位置之后,这个页面就是持久化的。然后这个页面就可以开始刷盘了,可以刷到真实的位置了,也许刷到的是 `ibata` 文件,也许是哪一个表的 `ibd` 文件中的某一个位置。但这里有一点需要注意,因为 Buffer Pool 中的页面,刷到真实文件时是异步 IO 的,那么只有当刷到自己表空间的刷盘操作完成后,两次写缓存数组的数据才可以被覆盖,或者说这个页面对应的两次写文件中的页面才能被覆盖,不然有可能造成这个两次写位置的页面被新的页面覆盖的问题。如果此时上次的真实表空间的刷盘没有完成,同时产生了页面断裂的问题,这样就出现了该页面不可恢复的问题,两次写的意义也就没有了。

这就是针对单一页面刷盘时两次写机制的实现,而在新版本 MySQL 5.7 中,还加入了针对 Buffer Pool 批量刷盘的两次写实现方式,实现方式在下面介绍。

批量页面刷盘

很明显,单一刷盘情况下开启了两次写,IO 次数的增加导致性能差很多,但 InnoDB 还有另外一种刷盘方式叫作批量刷盘。在考虑到性能的情况下,批量刷盘当然不能使用单一页面刷盘的方式,因为页面刷盘操作本身是批量的,如果两次写仍使用单一页面方式的话,性能并不会有所改善,所以 MySQL 5.7 采用了一种新的两次写方式。

MySQL 5.7 的实现方式新增了一个文件,文件路径及名称可以通过参数 `innodb_parallel_doublewrite_path` 来控制。启动数据库时,如果两次写文件不存在,那么这个参数可以指定绝对路径的两次写文件,也可以只指定文件名使文件被创建到 `datadir` 目录下。

上面已经提到,这种情况下是批量刷盘的,而批量刷盘包括两种方式,分别是 LRU 方式和 LIST 方式。这两种方式其实就是我们所熟知的两种页面刷盘的算法,当 Buffer Pool 空间不足时,再载入新的页面就必须要将一些不怎么用到的、老的页面淘汰出去,此时系统就会从

LRU 链表中找到最老的那些页面，进行批量刷盘，将空间还回到空闲空间中去，这种情况就是 LRU 刷盘；而当日志空间不足，或者后台 Master 线程在定时刷盘时，不需要区分页面的新旧状态，只需要选择 LSN 最小的那些页面，从前到后刷一批页面到文件中，此时所用的策略就是 LIST 方式。这两种方式的刷盘性质不同，时机不同，所以在两次写机制中，就区分开来了。下面来详细讲述批量页面刷盘的实现方式。

两次写组织结构

批量的两次写，是按照刷盘算法来区分的，包括 LIST 及 LRU。在两次写中，这两种刷盘方法对应的两次写空间互不干涉。同时，InnoDB 自身的整个 Buffer Pool 分为多个 Instance，每一个 Instance 管理自身的一套两次写空间，而针对每一个 Instance 的每一个刷盘方法的批量缓存空间大小，是通过参数 `innodb_doublewrite_batch_size` 来控制的，默认值为 120。这样算下来，`innodb_parallel_doublewrite_path` 所指的文件大小的计算方法如下。

$$\begin{aligned} \text{两次写文件页面个数} = & \text{innodb_buffer_pool_instances} * 2(\text{LIST} + \text{LRU}) \\ & * \text{innodb_doublewrite_batch_size} \end{aligned}$$

有兴趣的同学可以去检验一下算出来的值是不是真实的文件大小。

图 10.1 所示为批量刷盘时，两次写缓存空间的组织结构。

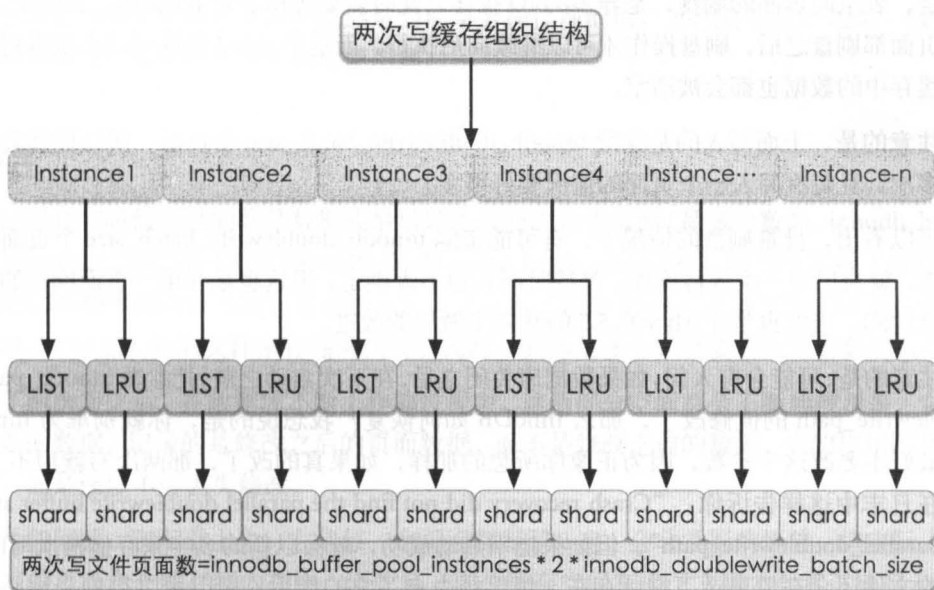


图 10.1

图 10.1 中落到最终的每一个 shard, 其实就是一个 batch, 对应的参数就是 `innodb_doublewrite_batch_size`。一个 shard, 有一个数组, 长度为 `innodb_doublewrite_batch_size`, 与单一页面刷盘的两次写是一样的, 只是这个数组只属于一个 shard 而已。

批量刷盘两次写实现原理

在了解了批量刷盘两次写的设计方案之后, 就可以了解它的刷盘过程了。假设由于页面淘汰, 有些页面必须要刷出磁盘, 系统要做一次批量刷盘操作, 这次就是 LRU 方式的, 那么此时系统就需要将当前页面加入到两次写缓存中, 根据当前页面所在的 Instance 号及刷盘类型就可以找到对应的 shard 缓存, 找到之后, 判断当前 shard 是否已经满了, 即是否已经达到了 `innodb_doublewrite_batch_size` 的大小, 如果没有达到, 则将当前页面内容追加复制到当前 shard 缓存中, 这样当前页面的刷盘操作就完成了。这里并不像单一页面那样, 先写入两次写缓存空间, 然后写入 `ibdata` 文件的两次写空间, 最后还需要立即将页面的真实内容刷入表空间, 对于批量刷盘来说, 只需要写入 shard 缓存即可。

如果当前 shard 中缓存的页面个数已经达到了 `innodb_doublewrite_batch_size`, 则说明当前缓存空间已经满了, 此时不得不将当前 shard 缓存的页面写入两次写文件中, 写完之后再写两次写文件 FLUSH 到磁盘, 最后将对应的真实页面刷盘, 但此时可能就是随机写入了, 因为在两次写缓存中虽然是连续的, 但对应的真实页面就不会是这样了。这里还需要注意的一点就是, 表空间页面的刷盘, 是异步的 IO 操作, 此时需要等待异步 IO 完成, 且整个 shard 中的页面都刷盘之后, 刷盘操作才可以继续向后执行, 而这个 shard 也就可以再次重新使用了, 缓存中的数据也都会被清空。

需要注意的是, 上面写入的是连续 `innodb_doublewrite_batch_size` 个页面, 所以性能会比写这么多次, 且每次写入一个页面的情况要好很多。

现在可以看出, 批量刷盘的情况下, 有可能每隔 `innodb_doublewrite_batch_size` 个页面的刷盘操作, 就会出现一次等待操作, 等待时间长短不太确定, 但这也是在单一页面刷盘的基础上优化过的, 当然也是在 MySQL 5.5 的基础上做出的改进。

需要注意的是, 可能会有人问, 如果数据库关闭之后, 在下次启动之前, 把参数 `innodb_parallel_doublewrite_path` 的值修改了, 那么 InnoDB 如何恢复? 我想说的是, 你就别难为 InnoDB 了, 最好不要改这个参数, 因为正像你所想的那样, 如果真的改了, 那两次写就用不上了, 只会在日志中这样告诉你: “Crash recovery did not find the parallel doublewrite buffer at `innodb_parallel_doublewrite_path`”。但如果侥幸启动成功, 就会以新的名字来存储两次写内容, 之前的文件还会在原目录下继续存在, 但已经失去了它的作用, 所以可以很安全放心地将其删除。

两次写的作用

上面已经讲完了两种两次写的实现原理，最后说一下它是如何起作用的。

在数据库启动时（异常关闭的情况下），都会做数据库恢复（redo）操作。在恢复的过程中，数据库都会检查页面是不是合法（校验等），如果发现一个页面的校验结果不一致，则此时会用到两次写机制，用两次写空间中的数据来恢复异常页面的数据，这也正是为了处理这样的错误而设计的。

此时的操作已经很明白了，将两次写的 2 个簇都读出来，再将 `innodb_parallel_doublewrite_path` 文件的内容读出来，然后将所有这些页面写回到对应的页面中去，这样就可以保证这些页面是正确的，并且是在写入前已经更新过的（最新数据）。在写回对应页面中去之后，就可以在此基础上继续做数据库恢复，且不会再遇到这样的问题了，因为最后有可能产生写断裂的数据页面都已经恢复了。

发散思维

上面所讲的都是数据页面有问题的情况下可以通过两次写页面来恢复，但是如果两次写页面本身发生写断裂怎么办呢？对于这个问题，其实不必担心，因为如果两次写有问题，则数据页面本身就不会做写操作，此时系统挂了，发生错误的是两次写页面，而数据页面在挂之前都是在 `buffer` 里面，文件中依然是当前事务操作前的值，并没有变化，还是一致状态，这意味着两次写页面根本就不会被使用到。

总结

两次写在任何时候记录的都是数据库最后发生改变的若干页面（最多个数为：`innodb_buffer_pool_instances * 2 * (LIST + LRU) * innodb_doublewrite_batch_size + 128`），这与之前所了解的 2MB（MySQL 5.6 版本的设计方案）是有出入的，之前所了解的 2MB 只是这里的 128 个页面，在 MySQL 5.7 中加入新的文件来存储两次写页面时，具体是多少，就要看配置参数的大小了。

开启了两次写之后，在数据库不断工作的过程中，这两部分空间都会不断地被覆盖，它始终是最新的数据，记录的是修改之后的页面数据，而不是修改之前的数据，它的作用不是还原数据，而是保证不会丢失修改。

至于性能问题，表面看上去，它是每一个页面都写了 2 遍，会非常影响性能，但实际上，由于所写的页面会先缓存到内存中，因此每一部分缓存空间在满了之后才会真正地写入文件。而对于磁盘而言，顺序写与随机写（每个页面自己写）的性能是相差很大的，两次写正是将若干数量的页面组合起来形成连续的空间写入两次写空间中，有效地利用了这个特点，所以性能不会相差 1 倍。实际上，经过测试，两次写使性能降低了约 10%。当然，这是针对普

通的机械磁盘而言的，对目前比较流行的 SSD 来说，随机写已经不是问题，性能影响可能更小。

针对这一问题，目前有些硬件存储厂商已经和 MySQL 合作，对两次写做了一些更彻底的优化。通俗来说，两次写其实不是什么优点、特性等，它只是一个被动解决问题的方案而已，对性能肯定是有影响的。这个问题的本质就是磁盘在写入时，都是以 512 字节为单位，不能保证 MySQL 数据页面 16KB 的一次性原子写入，所以才有可能产生页面断裂的问题。而目前有些厂商从硬件驱动层面做了优化，可以保证 16KB（或者其他配置）数据的原子性写入，如果真是这样，那两次写就完全没有必要了，取消两次写，才是最终级的优化，这值得我们期待。相信总有一天，两次写会被逐渐淡忘，成为过去，成为历史，不再被人们津津乐道。

其他一些数据库完全没有类似两次写的问题，比如达梦等，这主要是由于它们采用了全物理的 REDO，将一个页面的写操作都拆成一个个小的物理写入，这种情况下就不会存在写断裂的情况。因为不管怎么写，日志都是对一个页面操作的重演，在 REDO 做完之后，页面的状态肯定是正确的。

InnoDB 日志管理机制

InnoDB 存储引擎是支持事务 ACID 特性的,它是以二十多年前 IBM 的一篇著名文章 *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging* 为理论基础,大多数关系型数据库的实现都是基于这个理论的,包括 Oracle、DM 等。

这个理论基本就是一个关系型数据库相关的数据库恢复原型设计,包括日志、回滚、REDO、并发控制、Buffer Pool 管理等方面,内容非常全面。同时,这些内容是一个不可分割的整体,它们的共同目标之一就是保证数据库数据的一致性,保证数据库事务的 ACID 特性,所以这一章要讲的东西都是互相迁连的,它们之间相互作用,相互配合,相互驱动,才能保证数据库的数据完整性。下面就先从 Buffer Pool 的实现开始讲起。

InnoDB Buffer Pool

Buffer Pool 的背景

InnoDB 的 Buffer Pool 主要用来存储访问过的数据页面,它就是一块连续的内存,通过一定的算法可以使这块内存得到有效的管理。它是数据库系统中拥有最大块内存的系统模块。

InnoDB 存储引擎中数据的访问是按照页(有的也叫块,默认为 16KB)的方式从数据库文件读取到 Buffer Pool 中的,然后在内存中用同样大小的内存空间来做一个映射。为了提高数据访问效率,数据库系统预先就分配了很多这样的空间,用来与文件中的数据进行交换。访问时按照最近最少使用(LRU)算法来实现 Buffer Pool 页面的管理,经常访问的页面在最前

面,最不经常的页面在最后面。如果 Buffer Pool 中没有空闲的页面来做文件数据的映射,就找到 Buffer Pool 中最后面且不使用的位置,将其淘汰,然后用来映射新数据文件页面,同时将它移到 LRU 链表中的最前面。这样就能保证经常访问的页面在没有刷盘的情况下始终在 Buffer Pool 中,从而保证了数据库的访问效率。

Buffer Pool 的大小可以在配置文件中配置,由参数 `innodb_buffer_pool_size` 的大小来决定,默认大小为 128MB。在 MySQL 5.7.4 之前,一旦 MySQL 已经启动,这个值便不能再做修改,如果需要修改,只能退出 MySQL 进程,然后修改对应的配置文件来设置新的 Buffer Pool 大小,重新启动后才能生效。这在运维上非常不方便,因为很多时候,需要去调整 Buffer Pool 的大小,特别是在单机多实例,或者提供云数据库服务的情况下,我们需要根据用户及实际业务的需要,不断地去动态增加或减少 Buffer Pool size,从而合理地利用内存及优化数据库。

让人庆幸的是,MySQL 官方也发现了这种不便。在 MySQL 5.7.5 之后,MySQL 在源码上改变了对 Buffer Pool 的管理,可以在 MySQL 进程运行的情况下,动态地配置 `innodb_buffer_pool_size`。另外,需要强调的是,如果 Buffer Pool 的大小超过了 1GB,应该通过调整 `innodb_buffer_pool_instances=N`,把它分成若干个 instance 的做法,来提升 MySQL 处理请求的并发能力,因为 Buffer Pool 是通过链表的方式来管理页面的,同时为了保护页面,需要在存取的时候对链表加锁,在多线程的情况下,并发去读写 Buffer Pool 里面缓存的页面需要锁的竞争 and 等待。所以,修改为多个 instance,每个 instance 各自管理自己的内存和链表,可以提升效率。

Buffer Pool 实现原理

在启动 MySQL 服务时,会将所有的内嵌存储引擎启动,包括 InnoDB。InnoDB 会通过函数 `buf_pool_init` 初始化所有的子系统,其中就包括了 InnoDB Buffer Pool 子系统。Buffer Pool 可以有多个实例,可以通过配置文件中的参数 `innodb_buffer_pool_instances` 来设置,默认值为 1,实现多实例的 Buffer Pool 主要是为了提高数据页访问时的并发度。每个实例的空间大小都是相同的,也就是说系统会将整个配置的 Buffer Pool 大小按实例个数平分,然后每个实例各自进行初始化操作。

在代码中,一个 Buffer Pool 实例用 `buf_pool_t` 结构体来描述,这个结构体是用来管理 Buffer Pool 实例的一个核心工具,它包括了很多信息,主要有如下四部分。- `FREE` 链表:用来存储这个实例中所有空闲的页面。- `flush_list` 链表:用来存储所有被修改过且需要刷到文件中的页面。- `mutex`:主要用来保护这个 Buffer Pool 实例,因为一个实例只能由一个线程访问。- `chunks`:指向这个 Buffer Pool 实例中第一个真正内存页面的首地址,页面都是连续存储,所以通过这个指针就可以直接访问所有的其他页面。

上面的两个链表,管理的对象是结构体 `buf_page_t`,这是一个物理页面在内存中的管理结构,是一个页面状态信息的结合体,其中包括所属表空间、Page ID、最新及最早被修改的 LSN 值

(最早 LSN 信息会在做检查点时使用, 后面将会讲到), 以及形成 Page 链表的指针等逻辑信息。实际上, 这个结构是被另一个结构管理的, 它是 `buf_block_t`, `buf_block_t` 与 `buf_page_t` 是一一对应的, 都对应 Buffer Pool 中的一个 Page, 只是 `buf_page_t` 是逻辑的, 而 `buf_block_t` 包含一部分物理的概念, 比如这个页面的首地址指针 `frame` 等。关于 `buf_block_t`, 后面还会继续介绍。

初始化一个 Buffer Pool 实例内存空间的函数是 `buf_chunk_init`。一个 Buffer Pool 实例的内存分布是一块连续的内存空间, 这块内存空间中存储了两部分内容, 前面是这些数据缓存页面的控制头结构信息 (`buf_block_t` 结构), 每一个控制头信息管理一个物理页面, 这些控制头信息的存储, 占用了部分 Buffer Pool 空间, 所以在运维过程中, 看到状态参数 `innodb_buffer_pool_bytes_data` 总是比 `innodb_buffer_pool_size` 小, 就是因为控制头信息占用了部分空间。实际的分配方式是, Buffer Pool 页面从整个实例池中从后向前分配, 每次分配一个页面, 而控制结构是从前向后分配, 每次分配一个 `buf_block_t` 结构的大小, 直到相遇为止, 这样就将一个实例初始化好了。但一般情况下, 中间都会剩余一部分没有被使用, 因为剩余的空间不能再放得下一个控制结构与一个页面了。相应的分配代码如下。

```
/* 一个Buffer Pool的实例大小 */
chunk->mem_size = mem_size;

/* 申请对应大小的内存空间。这里虽然申请了, 但不能真正直接得
   到这部分空间, 而是通过mmap映射到相应大小的空间, 在后面真正使用
   到内存页面的时候, 才慢慢地逐渐分配真实的空间 */
chunk->mem = os_mem_alloc_large(&chunk->mem_size);

/* Allocate the block descriptors from the start of the memory block. */
chunk->blocks = (buf_block_t*) chunk->mem;

/* frame指向的是物理Buffer Pool页面, 所以需要以UNIV_PAGE_SIZE大小对齐 */
frame = (byte*) ut_align(chunk->mem, UNIV_PAGE_SIZE);

/* chunk的单位是Buffer Pool中的页面个数 */
chunk->size = chunk->mem_size / UNIV_PAGE_SIZE - (frame != chunk->mem);

/* Subtract the space needed for block descriptors.
   此处很重要, 这是为了给frame找到合适的位置。在整个chunk的空间中, 前面
   需要存储Buffer Pool页面的控制信息, 后面都是给Buffer Pool页面使用的。
   这里计算出这个Buffer Pool实例中可以存储多少个页面, 使得浪费的空间最少*/
{
```

```

uint size = chunk->size;
while (frame < (byte*)(chunk->blocks + size)) {
    /* 在这个循环中，frame从前向后，而chunk->blocks + size是从后向前，当第
       一次frame比blocks的最后一个大的时候，停止循环。那么，此时的frame就是
       当前Buffer Pool实例中第一个有效的Buffer Pool物理页面，
       而chunk->blocks + size是当前Buffer Pool中最后一个有效的页面控制信息。
       当然，这个是用来控制Buffer Pool中最后一个页面的 */
    frame += UNIV_PAGE_SIZE;
    /* 因为chunk->blocks的类型是buf_block_t，所以每次size变化时，
       chunk->blocks + size指针的值都增大sizeof(buf_block_t)，这也就是
       用来存储buf_block_t所需要的实际空间大小 */
    size--;
}
/* 循环停止，size为这个实例中最终的Page个数 */
chunk->size = size;
}

/* 从chunk最开始的位置存储blocks信息（页面控制信息） */
block = chunk->blocks;
for (i = chunk->size; i--;) {
    buf_block_init(buf_pool, block, frame);
    UNIV_MEM_INVALID(block->frame, UNIV_PAGE_SIZE);
    /* Add the block to the free list */
    UT_LIST_ADD_LAST(list, buf_pool->free, (&block->page));
    ut_d(block->page.in_free_list = TRUE);
    ut_ad(buf_pool_from_block(block) == buf_pool);
    /* 找到下一个block及下一个物理页面 */
    block++;
    frame += UNIV_PAGE_SIZE;
}

```

其中，`chunk->size`是在前面提前根据 Buffer Pool 实例内存大小计算出来的，可以存储的最多的 Page 及 Page 对应控制结构的个数。

一个 Buffer Pool 实例中的所有控制头信息连续存储在一起，所以控制信息存储完成之后才是真正的缓冲页面，图 11.1 所示的是一个 Buffer Pool 实例的内存分布情况。

对于 Buffer Pool 中的所有页面，都有一个控制头信息与它对应，从图 11.1 可以看出，每一个 `ctl` 都表示了一个属于自己的 `page` 使用情况。初始化实例时当然还需要对每一个控制头信息进行初始化，也就是每一个 `buf_block_t` 结构。初始化一个页面控制信息是通过 `buf_block_init` 函数实现的，`buf_block_t` 结构中包含了很多信息，主要包括如下四部分。

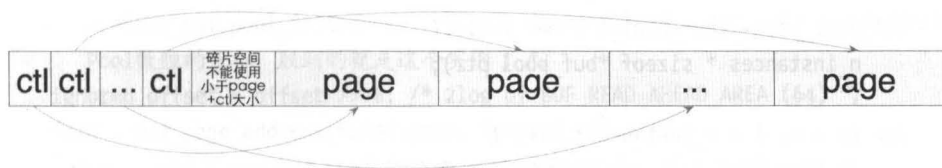


图 11.1

- 其对应的页面地址 frame。
- 页信息结构 `buf_page_t`，这个结构用来描述一个页面的信息，包括所属表空间的 ID 号、页面号、被修改时产生的 LSN (`newest_modification` 及 `oldest_modification`)、使用状态（现在共有 9 种状态）等（上面已经介绍过与 `buf_block_t` 的关系）。
- 用来保护这个页面的互斥量 `mutex`。
- 访问页面时对这个页面上的锁 `lock` (`read/write`) 等。

在初始化完每一个页面之后，需要将每一个页面加入到上面提到的空闲页链表中，因为这些页面现在的状态都是未使用 (`BUF_BLOCK_NOT_USED`)。

到现在为止，缓冲池的一个实例就算初始化完成了，在访问数据库的时候会通过这些内存页面来缓存文件数据。

相对于整个 Buffer Pool 而言，多个 Buffer Pool 实例之间的关系，需要在这里再讲述一下。上面所说的单个实例的初始化，是完全独立的，多个实例之间没有任何关系，单独申请、单独管理、单独刷盘，可以从其实现的代码中看到这一点，代码如下。

```
dberr_t
buf_pool_init(
    ulint    total_size, /*!< in: size of the total pool in bytes */
    ibool    populate,   /*!< in: virtual page preallocation */
    ulint    n_instances) /*!< in: number of instances */
{
    ulint    i;
    /* total_size表示的是参数innodb_buffer_pool_size的值，即总的Buffer Pool空间大小；
       n_instances表示的是innodb_buffer_pool_instances的大小。两者相除之后，
       就得到了每一个实例的大小 */
    const ulint size = total_size / n_instances;

    /* buf_pool_ptr用来管理整个Buffer Pool空间，不过只是一个数组而已。
       数组元素就是每一个Buffer Pool实例的管理首地址。buf_pool_ptr是系统
       全局变量，可以通过这个变量找到每一个Buffer Pool实例 */
}
```

```

buf_pool_ptr = (buf_pool_t*) mem_zalloc(
    n_instances * sizeof *buf_pool_ptr);

for (i = 0; i < n_instances; i++) {
    /* 根据下标得到每一个实例的Buffer Pool对象 */
    buf_pool_t* ptr = &buf_pool_ptr[i];

    /* 初始化每一个Buffer Pool实例, 就是上面介绍实例初始化代码中所讲述的内容 */
    if (buf_pool_init_instance(ptr, size, populate, i) != DB_SUCCESS) {

        /* Free all the instances created so far. */
        buf_pool_free(i);

        return(DB_ERROR);
    }
}

/* 统计最终的Buffer Pool空间大小 */
buf_pool_set_sizes();
buf_LRU_old_ratio_update(100 * 3 / 8, FALSE);

/* 初始化Buffer Pool自适应搜索系统 */
btr_search_sys_create(buf_pool_get_curr_size() / sizeof(void*) / 64);

return(DB_SUCCESS);
}

```

从代码中可以看出, 每一个 Buffer Pool 实例在整个 Buffer Pool 中确实是完全独立的, 而在具体使用时, 针对不同页面, 通过一个 HASH 算法, 来映射到一个具体的实例中, 对应的代码如下。

```

buf_pool_t*
buf_pool_get(
    ulint space, /*!< in: space id */
    ulint offset) /*!< in: offset of the page within space */
{
    ulint fold;
    ulint index;
    ulint ignored_offset;

```

/* 根据页面号offset进行计算, 再通过ignored_offset的值与表空间ID值得到fold, 然后


```

    再模srv_buf_pool_instances, 即Buffer Pool的实例数, 这样就得到了相对整个Buffer
    Pool数组的下标, 取到的就是这个实例了 */
    ignored_offset = offset >> 6; /* 2log of BUF_READ_AHEAD_AREA (64) */
    fold = buf_page_address_fold(space, ignored_offset);
    index = fold % srv_buf_pool_instances;
    return(&buf_pool_ptr[index]);
}

```

通过 Buffer Pool 多实例的管理机制, 可以减少系统运行过程中不同页面之间一些操作的相互影响, 从而很好地解决了由于页面之间的资源争抢导致的性能低下的问题, 所以在实际的运维过程中, 建议要分多实例的管理方式, 把 MySQL 及 InnoDB 用好, 让业务少一些烦恼。

REDO LOG 日志文件管理的用途

REDO LOG 是用来做数据库 crash recovery 的, 这是数据库保障数据安全的重要功能之一。在数据库操作中, 它保存了对 InnoDB 表中数据的修改记录, 所以也叫日志文件。在 InnoDB 存储引擎中, 一般默认包括 2 个日志文件, 新建数据库之后, 会有名为 ib_logfile0 和 ib_logfile1 的两个文件, 如果在启动数据库时, 这两个文件不存在, 则 InnoDB 会根据配置参数或默认值, 重新创建日志文件。

在 InnoDB 内部的日志管理中, 一个很重要的概念是 LSN, 全名叫 Log Sequence Number, 它用来精确记录日志位置信息, 且是连续增长的。在 InnoDB 中, 大小为 8 个字节的值, 它的增长量是根据一个 MTR (mini-transaction, 后面会讲到) 写入的日志量来计算的, 写多少日志 (单位字节), LSN 就增长多少。日志文件轮循一圈 (所有日志文件是以循环方式使用的), 那么 LSN 的增长量大约就是整个日志文件的大小 (日志文件存在文件头等会占用一部分空间)。它是一个集逻辑意义与物理意义于一身的概念。而在有些数据库中, LSN 是一个完全逻辑的概念, 每提交一个物理事务, LSN 就加 1。

上面提到, 日志文件是以类似循环圈的方式使用的, 如图 11.2 所示。

在 InnoDB 中, 通过日志组来管理日志文件, 是一个逻辑定义, 包含若干个日志文件, 一个组中的日志文件大小相等, 大小通过参数来设置。现在 InnoDB 只 持一个日志组。在 MySQL 5.5 及之前的版本中, 整个日志组的容量不能大于 4GB (实际上是 3.9GB 多, 因为还有一些文件头信息等), 到了 MySQL 5.6.3 版本之后, 整个日志组的容量可以设置得很大, 最大可以达到 512GB。

REDO 日志的写入, 都是字节连续的, 虽然看上去是多个日志文件, 但理解的时候, 完全可以把它想象成一个文件, 对每一文件掐头去尾, 把剩下的空间连接起来, 就是总的日志空间了。

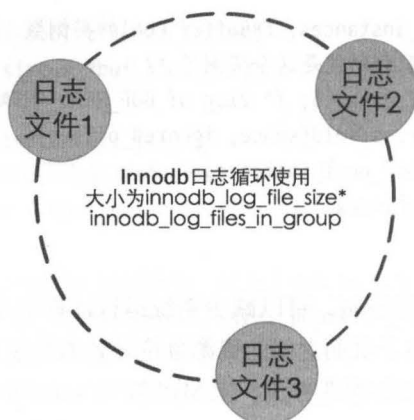


图 11.2

日志组中的每一个日志文件，都有自己的格式，内部也是按照大小相等的页面切割，但这里的页面大小是 512 个字节，由于历史的原因，考虑到机械硬盘的块大小是 512 字节，日志块大小也如此设计。这是因为写日志其实就是为了提高数据库写入吞吐量，如果每次写入是磁盘块大小的倍数，效率才是最高的，并且日志将逻辑事务对数据库的分散随机写入转化成了顺序的 512 字节整数倍数据的写入，这样就大大提高了数据库的效率。正是因为这个原因，REDO 日志才可以说是数据库管理系统与通过直接写文件来管理数据的最根本的区别之一。

下面展示的是日志文件的格式。

需要注意的是，图 11.3 中第一列指的是每一项在页面中的偏移位置，而下一项则是这个值加上该值在页面中所占长度得到的值。

图 11.3 中展示的 4 个页面（2048 字节），主要用于管理日志内容及整个数据库状态。在这 2KB 内容之后，就是正常的用来存储日志内容的部分，也是按照 512 字节页面大小的方式存储，图 11.4 中展示的是正常日志页面的格式。

普通页面中，都会有 12 个字节用来存储页面头信息，这些信息主要用于管理这个页面本身的数据存储方式。

- LOG_BLOCK_HDR_NO: 4 个字节，一个与 LSN 有关系的块号。
- LOG_BLOCK_HDR_DATA_LEN: 2 个字节，表示当前页面中存储的日志长度，这个值一般都等于 512-12（12 为页面头的大小），因为日志在相连的块中是连续存储的，中间不会存在空闲空间，所以如果这个长度不为 500，表示此时日志已经扫描完成（Crash Recovery 的工作）。
- LOG_BLOCK_FIRST_REC_GROUP: 2 个字节，表示在当前块中是不是有一个 MTR（关于这个概念的意义，会在下一节中专门介绍）的开始位置。因为一个 MTR 所产生的

日志量有可能是超过一个块大小的,那么如果一个 MTR 跨多个块时,这个值就表示了这个 MTR 的开始位置究竟是在哪一个块中。如果为 0,则表示当前块的日志都属于同一个 MTR;而如果其值大于 0 并且小于上面 LOG_BLOCK_HDR_DATA_LEN 所表示的值,则说明当前块中的日志是属于两个 MTR 的,后面 MTR 的开始位置就是 LOG_BLOCK_FIRST_REC_GROUP 所表示的位置。

第 0 页面	0	LOG_GROUP_ID
	4	LOG_FILE_START_LSN——文件 LSN
	12	LOG_FILE_NO——archived log file number
	16	LOG_FILE_WAS_CREATED_BY_HOT_BACKUP——归档
第 1 页面	0	LOG_CHECKPOINT_NO——检查点序号
	8	LOG_CHECKPOINT_LSN——检查点 lsn
	16	LOG_CHECKPOINT_OFFSET_LOW32——检查点文件偏移低 32
	20	LOG_CHECKPOINT_LOG_BUF_SIZE——log buffer size
	24	LOG_CHECKPOINT_ARCHIVED_LSN——归档
	32	LOG_CHECKPOINT_GROUP_ARRAY=LOG_CHECKPOINT_GROUP_ARRAY + LOG_MAX_N_GROUPS * 8=288 存储所有日志组的文件号及偏移,没用
	288	LOG_CHECKPOINT_CHECKSUM_1——校验
	292	LOG_CHECKPOINT_CHECKSUM_2——校验
	296	LOG_CHECKPOINT_FSP_FREE_LIMIT——废弃
	300	LOG_CHECKPOINT_FSP_MAGIC_N——废弃
	304	LOG_CHECKPOINT_OFFSET_HIGH32——检查点文件偏移高 32
第 2 页面	空着	
第 3 页面	同“第 1 页面”	

图 11.3

- LOG_BLOCK_CHECKPOINT_NO: 4 个字节, 存储的是检查点的序号。具体什么是检查点, 后面会详细介绍。

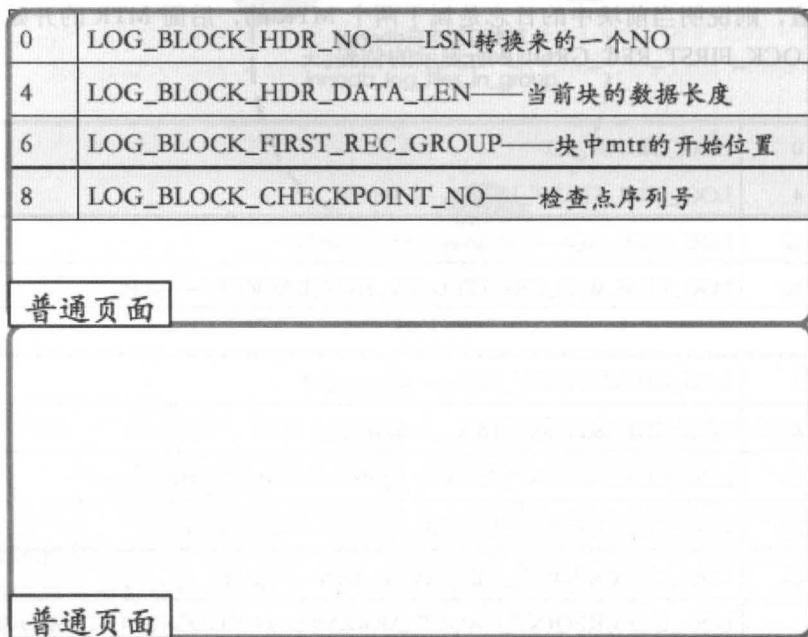


图 11.4

上面所讲述的就是日志文件的组织结构, 只有前面 2KB 是日志头, 后面所有的都是一个连续的、用来存储 MTR 产生的日志页面。

MTR InnoDB 物理事务

上面已经提到了关于 MTR 的概念, 实际上, 它是 InnoDB 存储引擎中一个很重要的用来保证物理页面写入操作完整性及持久性的机制。之所以被称为 MTR, 是因为它的意义相当于一个 Mini-transaction, 用 MTR 来表示, 这里把它称作“物理事务”, 这样叫是相对逻辑事务而言的。

对于逻辑事务, 熟悉数据库的人都很清楚, 它是数据库区别于文件系统最重要的特性之一, 它具有 ACID 四个特性, 用来保证数据库的完整性——要么都做修改, 要么什么都不做。物理事务从名字来看, 是物理的, 因为在 InnoDB 存储引擎中, 只要是涉及文件修改、文件读取等物理操作的, 都离不开这个物理事务, 可以说物理事务是 Buffer Pool 中的内存 Page 与文件之间的一个桥梁。

通过图 11.5 可以先了解一下 MTR 在 InnoDB 中的作用或意义。

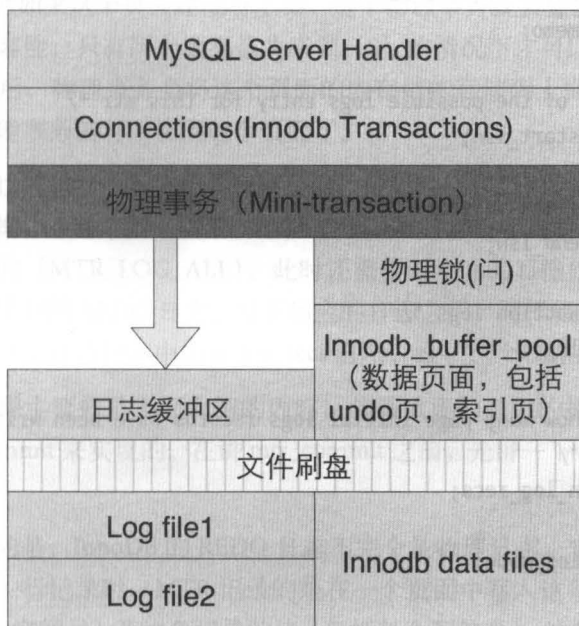


图 11.5

从图 11.5 中可以看出, 不管读还是写, 只要使用到底层 Buffer Pool 的页面, 都会使用到 MTR, 它是上面逻辑层与下面物理层的交互窗口, 同时也是用来保证下层物理数据正确性、完整性及持久性的机制。

前面已经介绍过 InnoDB 的页面 Buffer Pool 系统, 已经知道在访问一个文件页面的时候, 系统都会将要访问的页面载入到 Buffer Pool 中, 然后才可以访问这个页面, 此时可以读取或更新这个页面。在这个页面不断更新变化的过程中, 有一个系统一直扮演着很重要的角色, 那就是日志系统。因为 InnoDB 采用的也是 LOGWRITE-AHEAD, 所以所有的写操作, 都会有日志记录, 这样才能保证数据库事务的 ACID 特性。

而写日志是一个物理操作, 其实它也需要一个完整性。比如在底层页面插入一条记录, 如果只修改页头信息而没有修改页尾信息, 其实对于这个页面来说是不完整的, 所以这个物理操作还是需要一个机制来保证它的完整性的。那么在 InnoDB 中, 这个机制就是上面介绍的物理事务, 因为它也是用来保证完整性的, 所以也被称作“事务”。

物理事务既然被称为事务, 那它同样有事务的开始与提交, 物理事务的开始其实就是对物理事务结构体 `mtr_struct` 的初始化, 其中包括下面一些成员。

```

struct mtr_struct{
    /* memo stack for locks etc. */
    dyn_array_t memo;

    /* start lsn of the possible logs entry for this mtr */
    ib_uint64_t start_lsn;

    /* end lsn of the possible logs entry for this mtr */
    ib_uint64_t end_lsn;

    /* mini-transaction logs */
    dyn_array_t log;

    /* count of how many page initial logs records have been written to the
       mtr logs */
    ulint      n_log_recs;

    ulint      log_mode;
};

```

分别介绍一下每个成员的意义，如下。

- **memo**: 是一个动态数组空间，用来存储所有这个物理事务用到（访问）的页面（实际上存储的就是 Buffer Pool 中管理页面的控制块结构 `buf_block_t`），这些页面都是被所属的物理事务上了锁的（读锁或者写锁，某些时候会不上锁）。这个锁是读写锁、页面锁，与逻辑事务中所说的表锁和行锁要区分开来。
- **log**: 也是一个动态数组空间，用来存储这个物理事务在访问修改数据页面的过程中产生的所有日志，这个日志就是数据库中经常说到的重做（REDO）日志。
- **n_log_recs**: 表示这个物理事务产生的日志量，单位为日志记录条数。
- **log_mode**: 表示这个物理事务的日志模式，包括 `MTR_LOG_ALL`（写日志）、`MTR_LOG_NONE`（不写日志）等。
- **start_lsn**: 表示这个物理事务开始前的 LSN。
- **end_lsn**: 表示这个物理事务提交后产生的新的 LSN。

首先，在系统将一个页面载入 Buffer Pool 的时候，需要一个新开始（`mtr_start`）或者一个已经开始的物理事务，载入时需要指定页面的获取方式，比如是用来读取的还是用来修改的，这样会影响物理事务对这个页面上锁情况，如果用来修改，则上 X 锁，否则上 S 锁（当然还可以指定不上锁）。在确定了获取方式、页面的表空间 ID 及页面号之后，就可以通过函数 `buf_page_get` 来获取指定页面了，当找到相应页面后，物理事务就要对它上指定的锁，此时

需要对这个页面的上锁情况进行检查，一个页面的上锁情况是在结构体 `buf_block_struct` 的 `lock` 中体现的，此时如果这个页面还没有上锁，这个物理事务就会直接对其上锁，否则还需要考虑两个锁的兼容性，只有两个锁都是共享锁（S）的情况下才可以上锁成功，否则需要等待。当上锁成功后，物理事务会将这个页面的内存结构存储到上面提到的 `memo` 动态数组中，然后这个物理事务就可以访问这个页面了。

物理事务对页面的访问包括两种操作，一种是读，另一种是写。读就是简单读取其指定页面内偏移及长度的数据；写则是指定从某一偏移开始写入指定长度的新数据。同时，如果这个物理事务是写日志的（`MTR_LOG_ALL`），此时还需要对刚才的写操作记下日志，这里的日志就是逻辑事务中提到的 REDO 日志。写下相应的日志之后，同样将其存储到上面的 `log` 动态数组中，同时要将上面结构体中的 `n_log_recs` 自增，维护这个物理事务的日志计数值。

物理事务的读写过程主要就是上面介绍的内容，其最重要的是它的提交过程。物理事务的提交是通过 `mtr_commit` 来实现的。在讲 `mtr_commit` 之前，先讲一下，什么时候该提交，内部是如何控制的。

这里首先需要知道的是，InnoDB 的 REDO 日志不完全是物理日志，它包含了部分逻辑意义在里面，比如插入一行记录时，MTR 记录的是在一个页面中写入这条记录，内容大致包括页面号、文件号（表空间号）及这条记录的值（包括每个列信息），这样就有了逻辑概念。需要注意的是，在做 REDO 恢复时，需要保证这个页面是正确的、完整的，不然这个 REDO 就会失败，这也正是 InnoDB 存储引擎中著名的 `DOUBLEWRITE` 存在的意义，不过这是后话。而如果是纯物理的 REDO，日志内容应该会拆得更散，比如还是插入一条记录，它会记录页面号、文件号（表空间号）、页面内偏移值，并且有多个这样的 REDO 记录，因为会涉及多个位置的修改操作，这就没有任何逻辑内容了。而针对一个插入操作，需要在一个页面内的不同位置写入不同的数据，当然如果是纯物理 REDO，相应地会产生多条 REDO 记录，这是物理与逻辑的简单区别。

再说 MTR 的提交，一个逻辑事务是由多个物理事务组成，用来保证数据库的 ACID 特性的，有这个就够了，所以物理事务可以保证一次物理修改是完整的。所谓一次物理修改，可以理解为一个底层的相对完整的写入操作，比如插入一条记录的过程中，会包括写一条回滚记录及插入时写入一个页面等，那么这些逻辑上是一个动作的物理写入，就可以被认为是一个独立的物理事务，也就是在写回滚记录时执行 `mtr_start`，写完之后执行 `mtr_commit`，真正插入时写一个页面也是同样的道理。

接着介绍 MTR 提交的细节。物理事务的提交主要是将所有这个物理事务产生的日志写入到 InnoDB 日志系统的日志缓冲区中，然后等待 `srv_master_thread` 线程定时将日志系统的日志缓冲区中的日志数据刷到日志文件中，这会涉及日志刷盘时机的问题，不过还是先来看看 MTR、日志缓冲区及日志文件之间的关系，如图 11.6 所示。

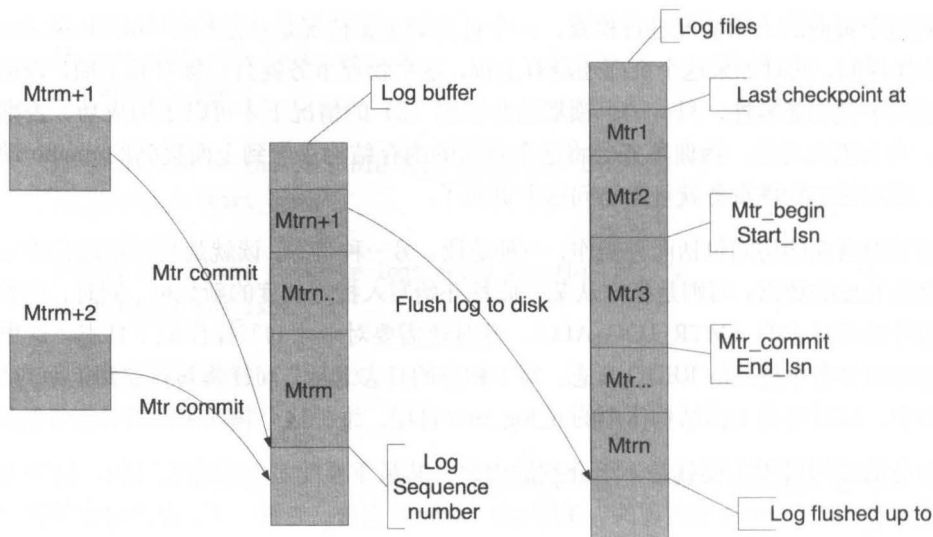


图 11.6

从图 11.6 中可以看出,左边的若干个 MTR 产生了各自的 REDO LOG,有些 MTR 已经提交了,有些正在写入。正在写入日志的 MTR,它们的日志都存储在自己 MTR 结构的 log 动态数组中,这个 MTR 还是不完全的,所以还是自己保存着,而对于那些已经提交的 MTR,它们对应的日志已经在提交的时候转存到了日志缓冲区中,相当于这些日志已经是实实在在地产生了,将来必然要占用数据库日志文件的一部分空间(除非数据库此时挂了)。

日志缓冲区的存储只是一个暂时的中间状态,日志缓冲区的大小可以通过参数 `innodb_log_buffer_size` 来设置,一般都比较小,存储不了太多的日志。因为已经提交并写入到日志缓冲的日志是确定的,所以它们是占用了 LSN 的,也就是说它们会使 LSN 变大。

最后提交的那个 MTR 代表着整个数据库最新的 LSN 值,也就是图 11.7 中所示的 Log Sequence number,这也正是在 MySQL 客户端中执行命令 `show engine innodb status\G` 时,返回的信息中 Log 模块中的第一行。

```
---
LOG
---
Log sequence number 81706062906725
Log flushed up to   81706062867689
Pages flushed up to 81705790802499
Last checkpoint at  81705784758349
```

图 11.7

而日志缓冲区也是有大小的,当多个 MTR 提交时,缓冲区被占满了,那么此时系统会将日志缓冲区的日志刷到日志文件中(这里涉及的另一个问题就是日志刷盘时机,这里只是一

种情况,其他的后面做专门介绍),为其他新的 MTR 释放空间。此时,日志的流向就是从中间的日志缓冲区向右边的日志文件转移,上面已经提到过,转移其实是平移,在缓冲区是什么内容,写入文件也是什么内容,也是完全连续的,且在日志文件中,还是一个一个的 MTR 连续存储。

最新写入日志文件的那个 MTR 产生的 LSN 值(图 11.7 中所示的 Log flushed up to),其实就是图 11.7 中所示的 Log 状态的第二行,也就是日志最新写入文件的 LSN 值,这个值的意义很重大,表示的是,到这个 LSN 为止,所有的修改都是完整的了,如果此时数据库挂了,写到这个位置的数据都是可以恢复的,而不需要去关心 Buffer 页面是不是被刷到磁盘。但此时在日志缓冲区中的日志所对应的操作就丢失了,这里是否会丢失事务数据与参数 `innodb_flush_log_at_trx_commit` 有关系,如果将参数 `innodb_flush_log_at_trx_commit` 设置为 1,当前事务的提交肯定会将日志缓冲区中的日志刷到日志文件中;如果设置为 2,那么日志只是写入了操作系统缓存,并没有写入磁盘,那么此时有可能丢失部分已经提交的事务,丢失多少由操作系统决定,这种情况下,即使数据库挂了,只要机器不挂,就问题不大,因为操作系统还会将它对应的缓存写入磁盘;但如果设置为 0 的话,就无能为力了,因为 InnoDB 只负责将事务对应的日志写入到日志缓冲区中,无论是操作系统,还是数据库,都不能保证日志的安全性,所以最好不要设置成这样。

进一步而言,日志文件的大小也是有限的,不可能无限量地将日志写入日志文件中。前面已经提到过,它是循环使用的,如果日志写入的头(图 11.7 中所示的 Log flushed up to)和尾相遇了,此时日志就不能再写入了,因为如果再写入的话,就要“追尾”了,这样会将之前产生的日志覆盖掉,导致日志不可用,不完整。此时就会使用一种机制来保证新的日志还能继续写入,尾部日志还是完整的,这个机制叫作 checkpoint(检查点)。

说白了,日志产生的作用,是将随机页面的写入变成顺序日志的写入,从而用一个速度更快的写入来保证速度较慢的写入的完整性,以提高整体数据库的性能。其根本目的是要将随机变成顺序,所以日志的量才是一个相对固定循环使用的空间。有了这个思想之后,使用检查点来保证日志的重复写入、数据库完整性就是顺其自然的事情。

使用检查点来保证数据库完整性的主体思想,主要是让日志失效,也就是让 Buffer Pool 中的页面修改写入到磁盘上面。因为日志的存在实际上就是让 Buffer Pool 中的 Page 尽可能少地刷磁盘,尽可能长时间地将页面数据缓存起来,尽可能提高访问速度,因为不管如何修改,Buffer Pool 中的页面都是最新的,只是不一定写入磁盘中(没有刷入没关系,由日志来保证)。如果日志文件大小不够用,此时只要将 Buffer Pool 中的某些页面刷入到磁盘中,其对应的日志就失效了,因为这些日志就是用来保证 Page 没有刷入时但数据库挂了的情况下数据库的完整性的,而这些 Page 如果已经写入磁盘了,相应的日志也就没有用了,这就是检查点的根本意义所在。

而上面提到的,做检查点时,只是将某些页面刷入磁盘,其中的“某些”是有讲究的。俗话说:“家有三件事,先从紧处来”,现在的问题是日志空间不够用了,而日志是循环使用的,必须是按照顺序,不能跳着写,所以最主要的是从 LSN 值最小的日志开始,按照从小到大的顺序不断地让这些日志失效。每次做检查点都会有一个比例,此时系统会根据最小的有效 LSN (`min_valid_lsn`) 和检查点处理的日志比例计算出最大的将要失效的 LSN 值(取名叫 `lsn_checkpoint_up_to`)。计算完之后,再去扫描 Buffer Pool 的 `flush_list` 链表,找出所有被更新过的页面中,曾经修改这些页面的 MTR 对应的 LSN 中的最小值(因为一个页面有可能被多次修改,但只需要考虑最小的 LSN 的那一次,使用的是前面介绍结构 `buf_block_t` 时,这里面所存储的 `oldest_modification` 的值),如果这个值比 `lsn_checkpoint_up_to` 值小,就将这个页面刷入磁盘,也就是说,如果将小于 `lsn_checkpoint_up_to` 的 MTR 修改过的页面都刷入磁盘了,那么日志文件中在 LSN 值 `lsn_checkpoint_up_to` 以前的日志就都可以失效了,那么在整个日志文件空间中,从 `min_valid_lsn` 到 `lsn_checkpoint_up_to` 之间的空间,又可以被重新使用了,直接覆盖即可,而不会导致数据库不完整、数据丢失等问题。

上面讲的整个检查点过程,用一个更形象的图表示,如图 11.8 所示。

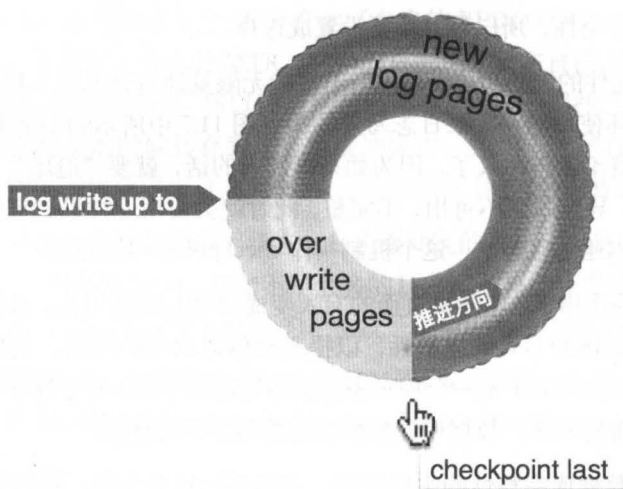


图 11.8

此时,再接着上面 MTR 产生日志的图 11.6 来讲,上面找到的日志文件的位置 `lsn_checkpoint_up_to` 就是图 11.7 中所示的 Last checkpoint at,也是上面命令 `show engine innodb status\G` 中关于 Log 部分的第四行信息。而从这个点开始到最新的已经刷盘的日志文件位置 `Log flushed up to` 之间的日志都是有效日志了,不能被覆盖,只有空间又不够用了的情况下,再将最小的有效日志位置向前推,产生新的位置,像这样不断循环,周而复始的工作,这就是日志、Buffer Pool 及检查点之间的工作原理。

上面提到的 `show engine innodb status\G` 命令生成的 Log 部分中显示的第三行信息，是 Buffer Pool 中 Page 刷盘时刷到的一个最新的 LSN。但此时检查点的最新点不一定做得及时，所以它是大于等于第四行的，而图 11.7 中所示的四行对应的值，从上到下以递减的顺序排序，其中的道理都已经非常明确了。

上面已经讲过，物理事务和逻辑事务一样，也是可以保证数据库操作的完整性的。一般说来，一个操作必须要在一个物理事务中完成，也就是说要么这个操作已经完成，要么什么也没有做，否则就有可能造成数据不完整的问题，因为在数据库系统做 REDO 操作时是以一个物理事务为单位做的，如果一个物理事务的日志是不完整的，则它对应的所有日志都不会重做。那么，如何辨别一个物理事务是否完整呢？这个问题是在物理事务提交时用了一个很巧妙的方法来保证的。在提交前，如果发现这个物理事务有日志，则在日志最后再写一些特殊的日志，这些特殊的日志就是一个物理事务结束的标志，提交时一起将这些特殊的日志写入，在重做时如果当前这一批日志信息最后面存在这个标志，则说明这些日志是完整的，否则就是不完整的，就不会重做。

物理事务提交时还有一项很重要的工作就是处理上面结构体中动态数组 `memo` 中的内容，现在已经知道这个数组中存储的是这个物理事务访问过的所有页面，并且都已经上了锁。在它提交时，如果发现这些页面中已经有被修改过的，这些页面就成了脏页，这些脏页需要被加入到 InnoDB Buffer Pool 中的更新链表中（讲 BUFFER 时已经讲过）。当然，如果已经在更新链中，则直接跳过（不能重复加入），`svr_master_thread` 线程会定时检查这个链表，将一定数目的脏页刷到磁盘中，加入之后还需要将这个页面上的锁释放掉，表示这个页面已经处理完成；如果页面没有被修改，或者只是用来读取数据的，则只需要直接将其共享锁（S 锁）释放掉即可。

上面的内容就是物理事务的一个完整的讲述，它是比较底层的一个模块，牵扯的东西比较多，这里重点讲述了物理事务的意义、操作原理、与 BUFFER 系统的关联、日志的产生等内容。

下面继续讲述有关日志的其他内容。

日志的意义

上面已经讲述过，日志是在逻辑事务对数据库做 DML 操作时，其所包含的物理事务 MTR 所记录的，针对所有涉及的 Buffer Pool 页面的修改记录。

为了更好的讲述日志的意义，这里通过以下几个方面来更好地说明。

假如没有写日志

假如没有写日志,那数据库在做了任何修改之后,必须要直接将 Buffer Page 刷磁盘,不然如果此时数据库挂了,即使事务已经被提交,这些修改还是没办法恢复。这将带来的灾难是,IO 大量增加。此时的数据库,相当于是一个简单的文件系统,无论写什么数据,都必须马上刷入磁盘,Buffer Pool 的作用可能只是一个用来修改文件页面的临时缓存而已。

假如没有写日志,在数据库做了 DML 操作之后,数据库可能在事务没有提交时就将 Buffer Page 刷到磁盘了,但此时需要回滚。而我们知道,回滚段的内容也是通过 Buffer Pool 管理的,它的每个页和 B 树页面是一样的,只是作用不一样而已。由此可知,回滚段数据也是通过 REDO 日志来保证完整性的。那么如果没有了日志,Buffer Page 中的回滚段页面也需要直接写入,没有了任何缓存,性能就会非常低。

假如没有写日志,数据库在关闭(挂掉)后再启动时,就不需要做 REDO 操作了(因为没有写日志),但需要做 UNDO 操作,因为 UNDO 不是通过 REDO 来恢复的,而是自己写入(假设每次写 Buffer Page 之后都直接刷盘了),所以回滚段是有效的,还可以让没有提交的事务回滚掉(因为如果一个事务修改的页面很多的话,肯定会有一部分页面先被刷掉,所以有可能需要回滚),勉强还可以保证数据库的完整性。

综合上面的假设,现在已经明白,日志的作用就是用来保证 Buffer Pool 页面的数据写入不丢失。反过来说,如果每个 Buffer Pool 中的 Page 每次都刷入到磁盘中,这样就不需要 REDO 日志了,此时的数据库就成了一个文件系统,因为 Buffer Pool 每次都进行刷盘,相当于每次写完直接写文件。所以说,日志是数据库管理系统与文件系统最核心的区别。

所以如果没有日志,数据库的性能就低到完全没有办法用了。因为 IO 太大了,同时,这种 IO 操作都是随机写入,很容易导致 IO 到达瓶颈,所以为了提高数据库性能,就必须使用 REDO 日志机制。

使用日志能提高性能的关键原因,有以下三个方面。

- 因为日志是用来记录 Buffer Pool 中 Page 的修改记录的,所以把对 Page 的写入转化为对日志的写入,那此时 Page 就不需要每次都刷盘,写 Page 页面只需要在内存中写入即可,性能会非常好。
- 通常,一个页面是 16KB,如果不写日志的话,每次的写入单位还是 16KB,即使修改很少量的数据,也是如此。这样会导致无效 IO 非常严重,反过来说,也只有通过日志机制,才能真正体现出真实写入的数据量,不会存在对 IO 的浪费,Page 的刷盘数量会大大减少。
- 如果没有日志,就会每次都刷 Page,而这些 Page 的相对位置是乱的,并不是顺序的,刷盘大多都是随机 IO,这对于机械硬盘来说,性能是非常差的,而有了日志,就可以巧妙地将随机 IO 转化成日志的顺序 IO,这将大大地提高 IOPS,性能也会非常好。

日志文件大小的区别

使用日志对数据库的性能有很大的影响，那对于日志来说，还有什么其他因素会影响数据库的性能呢？那就是日志文件空间容量。

现在已经知道，日志在设置好后其容量是固定的，它是循环使用的，如果不够用了，引发的事件是做一个检查点，让最小有效的 LSN 向前推，让出一部分空间给新产生的日志来使用。也就是说，只要这个日志空间未用完，那么 Buffer Pool 中的 Page 就会一直不刷盘（因为还有其他的刷盘时机，所以这里单指因为日志不够用导致检查点的刷盘），任何修改都是在内存中发生的。那么，下面做一个计算。

假如当前日志容量设置为 128MB，某一个 DML 操作只针对某一行记录一直做修改操作。每次操作产生日志量为 1KB（包括 Buffer 中数据页面的修改及 UNDO 记录的产生），这样算下来，128MB 的日志量可以容纳对这条记录的 131072（128MB/1KB）次修改。也就是说，在这么多次修改之后，这个页面才需要刷盘，才会产生一次随机刷盘操作。而如果把日志文件设置为 1280MB，很容易知道，这将容纳对这条记录的 1310720 次修改，这么多次修改只产生一次随机刷盘操作。而如果还是 128MB 的话，则需要 10 次随机刷盘。很明显，日志容量对数据库的性能还是有很大影响的。

但也不是设置得越大越好，这里有以下两点需要注意。

- 如果设置得非常大，固然性能可能会很好，但如果某一天（真的有可能到来），数据库异常挂了，此时可能有很多的日志都没有刷盘，也就是 Log flushed up to 与 Last checkpoint at 两个值之间相差太多，恢复起来可能需要比较长的时间。但这个一般问题不大，本身挂的概率不大，同时 REDO 日志的恢复是顺序的，都是根据页面号的大小排序恢复的，所以比较快。同时，在以后的 MySQL 版本中，会有多线程 REDO 恢复（听说的），这样就更快了，所以这一点不需要太担心。
- 日志容量大小的设置，最好与 Buffer Pool 的总大小匹配。如果日志容量太小，Buffer Pool 太大，这就会导致 Buffer Pool 频繁做检查点，大的 Buffer Pool 不能被好好利用。如果是日志容量很大，而 Buffer Pool 很小，此时 Buffer Page 经常会被淘汰出去，增加了 IO 频次，同时如果数据库意外挂掉，Buffer Pool 小的话，恢复起来也会比较慢。一般情况下，Buffer Pool 的总大小与日志容量的大小比例最好保持在 10~5:1 的范围内。

日志记录格式

前面已经讲述了太多日志相关的内容了，这一节将要讲一下具体到一个日志记录时它是如何组织的，一条日志记录究竟存储了什么。这些问题在这一节都会说清楚。

前面已经讲到, InnoDB 的日志是具有逻辑意义的物理日志, 所以, 日志记录的格式就不完全是物理信息, 而是有一定逻辑意义的。首先看一个基本的格式, 如图 11.9 所示。

Type	spaceid	offset	data
------	---------	--------	------

图 11.9

图 11.9 中各列代表的意义如下。

- **Type**: 日志类型, 是一个日志记录的最高位, 只占一个字节的空間。
- **Space**: 表空间 ID 值, 如果是系统页面 (UNDO 页面, 或者是字典表页面), 则是 0; 如果是索引页面, 则是这个索引所在的表空间 ID 值。
- **Offset**: 上面 Space 所指定的文件中的页面号, 以页面大小为単位, 它是第几个页面 (从 0 开始计数), 则这个 Offset 就是几。
- **Data**: 表示这条日志记录对应的数据, 这个数据是不确定的, 根据不同的 Type 值而不同, 分别具有自己的格式。

Type 类型有很多, 下面列举了一些在 InnoDB 中比较常用的类型, 并简单做一些解释, 以便可以更好地理解。InnoDB 中的 REDO, 究竟是在做什么? 究竟存储了哪些内容? 功能是什么? 知道每个类型之后, 这些问题也就清楚了。



注意: 下面讲到的数据记录, 都是以 Compact 格式的记录为对象的, 其他类型这里不做考虑。

- **MLOG_1BYTE、MLOG_2BYTES、MLOG_4BYTES、MLOG_8BYTES**: 这四个类型, 表示要在某一个位置, 写入一个 (两个、四个、八个) 字节的内容, 在日志记录中, Type 分别是 MLOG_1BYTE (MLOG_2BYTES、MLOG_4BYTES、MLOG_8BYTES), Space 就是对应的表空间 ID, Offset 对应的是页面号。在 Data 中, 还需要存储三个 (四个、六个、十个) 字节, 前两个为要写入的数据在页面内的偏移值, 因为页面大小为 16KB, 所以需要两个字节来存储, 而后面才是真正需要写入的数据, 占一个 (两个、四个、八个) 字节, 这就是关于这个类型的日志的完整内容。
- **MLOG_WRITE_STRING**: 这种类型的日志, 其实和 MLOG_1BYTE 是类似的, 只是 MLOG_1BYTE 是要写一个固定长度的数据, 而 MLOG_WRITE_STRING 是要写一段变长的数据。Data 部分的格式, 首先用 2 个字节存储在页面中的写入位置, 然后是 2 个字节写入数据长度, 最后是存储指定长度的字符串。
- **MLOG_COMP_REC_MIN_MARK**: 这个类型的日志, 是在将一条记录设置为页面中的最小记录 (这个涉及页面管理的内容, 在一个页面中只有一个最小记录, 它指向的是 B

树下一层的最左边的节点)时产生的,因为只是打个标记,存储内容比较简单,除了基本的日志头外,在 Data 内容中只存储了这条最小记录在页面内的偏移位置。

- **MLOG_UNDO_INSERT**: 这个类型的日志,是用来保证一个插入操作可以在事务没有提交的情况下回滚时用的,在插入一条记录时,不止要写一个插入操作的日志(类型为 MLOG_REC_INSERT,后面会着重介绍),还要写一个针对这个操作的回滚记录。我们已经知道,回滚记录的写入,其实也是向 ibdata 文件中写入数据,同样也是写在 Buffer Pool 中的,这个操作对应的 REDO 日志,就是当前介绍的 MLOG_UNDO_INSERT 类型的日志。在数据库恢复时,只有这个 REDO 日志做完了,相应的 UNDO 记录才有效(存在),如果对应的事务没有提交,会通过这个回滚记录将这个插入操作回滚掉,这也正是 REDO 必须要在 UNDO 之前执行的原因。

至于这种类型的日志格式是什么样子的,与前面所说类型的区别还是在 Data 上面。前面两个字节存储的是回滚记录的长度,接着就是回滚记录的完整数据,不包括回滚记录前后各两个字节的指针信息,具体到回滚记录的格式,后面会讲述。

- **MLOG_INIT_FILE_PAGE**: 这个类型的日志比较简单,只有前面的基本头信息,没有 Data 部分。因为在 InnoDB 中,初始化一个页面,所有的信息都是固定的,没有额外的处理,只要表明初始化哪一个位置的页面就好了,所以没有 Data 部分。这里初始化页面所做的操作,只涉及对页面中文件管理方面的信息,比如这个页面的页面号、文件号(表空间 ID)等信息,与后面将要介绍的 MLOG_COMP_PAGE_CREATE 是不同的,这个属于页面管理的文件信息部分的初始化,而 MLOG_COMP_PAGE_CREATE 属于页面的索引、数据存储方面的管理信息的初始化。后者是在前者的基础上做的。
- **MLOG_COMP_PAGE_CREATE**: 这个类型的日志,其实和上面已经说过的 MLOG_INIT_FILE_PAGE 是一样的道理,因为在 Buffer 中创建一个新的可以使用的页面是固定的,只需要存储一个类型及要创建的页面的位置即可。创建一个页面所做的操作,包括初始化页面头信息,创建页面中的最小记录与最大记录,初始化页面中的记录数、HEAP 大小、HEAP 首地址及槽信息,初始化之后,这个页面就可以在 B 树中使用了,它是一个页面在没有插入任何数据时的状态。
- **MLOG_MULTI_REC_END**: 这个类型的日志是非常特殊的,它只起一个标记的作用,其存储的内容只有占一个字节的类型值。在前面介绍 MTR 时说到,一个 MTR 所写的日志,要么全部写入,要么全部不写入。那么,如何保证这个原子性就是通过这个类型的日志来实现的呢?即每次 MTR 提交时,都会在后面加上这个日志记录,用来表示这个 MTR 已经结束了。只有在恢复的时候才会使用它,在分析 MTR 时,只有找到这个日志,前面的日志才会去做 REDO,做完之后,再向后扫描找到这个日志,然后再去 REDO,如此反复。如果有一次找不到了,则说明日志文件是不完整的,已经扫描到的 REDO 日志就不会去执行了,从而保证了已经执行的 MTR 每个都是完整的。

- **MLOG_COMP_REC_CLUST_DELETE_MARK**: 这个类型的日志是表示, 需要将聚集索引中的某一个记录打上删除标志。因为, 众所周知, 在 InnoDB 数据库中聚簇索引的删除在没有提交之前, 只是打了一个删除标志而已。这个类型的日志记录内容, 除了基本的内容之外, 其 Data 数据的组成主要包括两个字节的索引列的个数 n , 以及两个字节的唯一索引的个数 u 。接下来存储的是所有索引列的长度信息, 每个列用 2 个字节存储, 占用空间 $2*n$ 个字节。然后, 再存储索引中两个系统列信息, 分别是 TRXID 在索引中的列位置信息、ROLLPTR 的值及 TRXID 的值。最后, 再存储当前要删除记录在所在页面中的偏移值, 也就是那条记录的头指针信息。这种类型的日志, 存储的内容比较复杂, 其 Data 部分使用图 11.10 来简明表示一下。

页内偏移	索引字段数	唯一键数	列数据长度1	列数据长度2	列数据长度n	TRXID列号	ROLLPTR	TRXID
------	-------	------	--------	--------	-------	--------	---------	---------	-------

图 11.10

这里有一个奇怪的地方是, 在给一个记录打删除标志时, 为什么不使用这条记录的主键值来直接定位, 而是使用了一些在定位记录时被认为是没有用的东西呢? 因为如果需要数据恢复, 只需要找到这行记录的主键信息, 就可以重新给这个记录打删除标志。那为什么存储的都是一些索引的定义信息呢? 比如索引列个数, 唯一键列个数, 每个列的长度等。关于这个问题, 是因为 InnoDB 还需要考虑其自身的问题, 那就是它的 REDO 日志是半逻辑半物理的, 在恢复时, 不能保证对应的数据字典是可用的 (因为数据字典的正确性还是需要 REDO 来保证), 所以这个日志就会记录一些索引的信息, 在恢复时使用这些信息来构造一个 LOG_DUMMY 表及 LOG_DUMMY 索引, 然后再用这个表和索引来辅助这个 REDO 日志的执行, 这样真正的表及索引可以不正确 (暂时的), 因为此时是不需要它们的。综合上面所述的日志 Data 部分, 就可以知道这条记录的确切信息了, 也就可以对它加删除标志了。

- **MLOG_COMP_REC_UPDATE_IN_PLACE**: 这个类型的日志, 和上面的 MLOG_COMP_REC_CLUST_DELETE_MARK 基本是差不多的, 只是此类型的日志会在最后存储原地更新后的记录信息, 包括所有被更新的列的信息。存储方式是: 前面用一个字节或两个字节来存储长度, 后面跟着的是更新后的数据, 直到记录所有的列为止。
- **MLOG_COMP_REC_DELETE**: 在 InnoDB 中, 删除数据是通过打删除标志来实现的, 但是, 在事务提交后做 Purge 操作时, 这条记录始终是要被删除的, 所以, 还存在一个真正将数据记录删除的操作, 那这个类型的日志就是用来记录这个动作的。不过这个日志需要记录的内容也比较少, 除了基本的日志头信息之外, 在 Data 中只需要存储这条记录在页面内的偏移即可。那么在恢复数据时, 这种类型的日志的作用就是会将这个页面中对应的记录直接删除掉, 而不再是打删除标记那么简单了。

- **MLOG_COMP_PAGE_REORGANIZE**: 这个类型的日志, 表示的是要重组指定的页面, 其记录的内容也是很简单的, 只需要存储要重组哪一个页面即可, 没有 Data 部分。在恢复的时候, 找到这个页面, 对其中的数据碎片做整理, 将页面内部的记录一条条向前移, 将原来记录之间不能再被使用的空间收回合在一起变成一块连续的空间, 这样原来貌似已经满了的页面, 又可以插入新的数据了, 这就是表的碎片整理过程。
- **MLOG_COMP_REC_INSERT**: 这个类型是在插入一条记录时产生的, 它的产生过程可能存在一点争议, 这里重点说一下。首先, 当然还是基本的日志头信息, 然后存储的是被插入记录在页面内的偏移信息, 接下来就是关于索引的信息, 这些都与上面 **MLOG_COMP_REC_CLUST_DELETE_MARK** 类型日志的内容相同。然而, 再往后所存储的信息就比较复杂了。首先会计算出当前要插入的记录与前一条记录第一个不相同的字节的位置, 然后在日志中记录从这个位置开始到当前记录结束位置之间的数据, 当然还有一些其他的信息, 比如第一个不相同的字节的位置信息等。这里主要想说的是它的设计方式, 简单一点说, 就是当前记录中存储的只是记录的后半部分数据, 前半部分数据依赖的是前一条记录, 这样存储会比存储整个记录省多少空间呢? 最主要的是, 这需要依赖插入数据之间的相关性, 如果非常像, 则可能会省一些, 否则可能效果不明显。

上面讲述的是, 在 InnoDB 存储引擎中, REDO 日志的一部分类型, 并对不同类型做了解释。从解释中可以看到, 基本上每一个类型其实都是具有逻辑意义的, 与 DML 相关的类型中, 不是存储了列数据, 就是存储了记录在页面内的偏移等信息, 这样做的优点有如下两点。

- 可以写 REDO 解析工具, 去做一个第三方的同步工具, 或者了解数据库做了什么操作, 类似 Binlog 内容, 但侧重点不同。
- 日志占用空间比全物理日志少。

最大的缺点就是系统首先要保证日志对应页面的正确性, 否则会造成逻辑日志执行不成功, 或者造成数据不一致等问题, 这个问题在 InnoDB 中的解决方式, 就是后面介绍的 Double Write 机制。

日志刷盘时机

前面已经介绍了大部分关于 REDO 日志的内容了, 但还有一个问题没有讲, 就是日志刷盘的时机, 也就是什么时候才会将日志刷入磁盘。

现在已经知道, 当 MTR 提交时, 所产生的日志, 都会先写入到 Log Buffer 中, 这是日志产生的最初来源。从这个源头开始, InnoDB 会在不同的时机, 将这些日志写入到磁盘, 分别有下面五种时机。

- Log Buffer 空间用完了，便会将已经产生的 Log Buffer 中的日志刷到磁盘中，这个时机在前面介绍 Mtr 时已经说过了。这是最普遍的一种方式。
- Master 线程在后台每秒钟刷一次，将当前 Log Buffer 中的日志刷到磁盘中。
- 每次执行 DML 操作时，都会主动检查日志空间是否足够，如果使用空间的量已经超过了预设的经验值，就会主动去刷日志，以保证在后面真正执行时，不会在执行过程中被动地等待刷盘，但这里只会是写文件（写入 OS 缓存中），不会刷磁盘。
- 在做检查点的时候，要保证所有要刷的数据页面中 LSN 值最小（最旧）的日志已经刷入到磁盘。不然，如果此时数据库挂了，日志不存在，但数据页面已经被修改，从而导致数据不一致，就违背了先写日志的原则。
- 提交逻辑事务时，会因为参数 `innodb_flush_log_at_trx_commit` 值的不同，产生不同的行为。如果设置为 0，则在事务提交时，根本不会去刷日志缓冲区，这种设置是最危险的，如果此时运气不好，那对数据库最新的修改都会丢失，即使事务已经提交了，但丢失的事务一般是最新 1 秒内产生的，因为 Master 线程会每隔 1 秒刷一次。如果设置为 1，则在事务提交时会将会日志缓冲区中的日志写入到文件中，同时会将这次写入强刷到磁盘中，保证数据完全不丢失，但这种设置会使得数据库性能下降很多，影响性能。如果设置为 2，则在事务提交时会将会日志写入到文件中，但不会去刷盘，只要操作系统不挂，即使数据库挂了，数据还是不会丢失，一般都是设置为 2 即可。关于这个问题，可以用图 11.11 来简单表示。

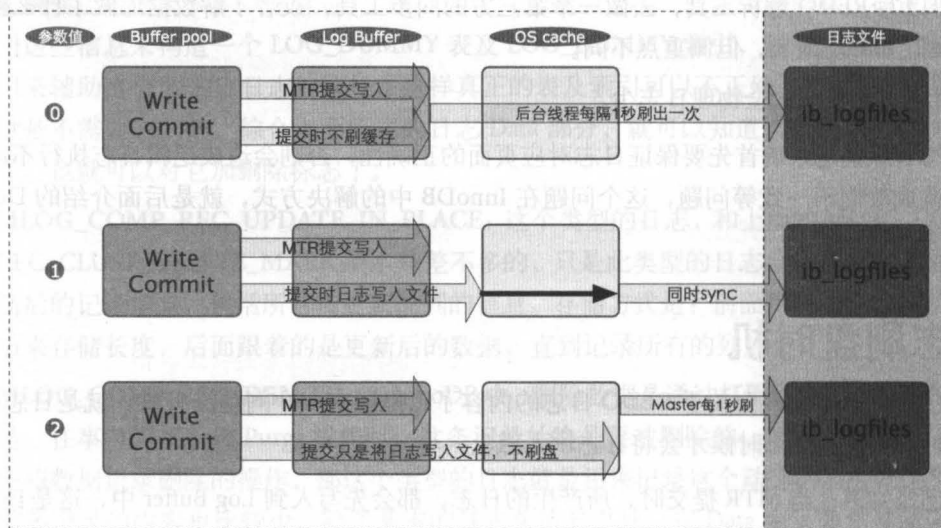


图 11.11

上面所说的基本上就是全部的日志刷盘时机了，相关内容都已经介绍清楚，在接下来的两节中，会重要讲述数据库的恢复问题。

REDO 日志恢复

前面已经很全面地介绍了日志的生成、格式、刷盘、工作原理等，但这些实际上只是数据库运行时的一个“累赘”，没办法才会这样做，因为如果数据库不挂，日志是没有用的，但不挂是不可能的，所以日志是必须要有的。而前面介绍的所有内容都是建立在有日志的前提下，解决如何提高性能，如何保证数据完整性等问题的。那这里将介绍关于日志的新内容，日志的用途之一：数据库恢复。

在第 5 章中，已经介绍了在 InnoDB 存储引擎的启动过程中，InnoDB 需要做的事情有哪些，具体细节可以参考第 5 章了解。在这一节中，需要重点关注的主要有两个，包括 `recv_recovery_from_checkpoint_start` 及 `recv_recovery_from_checkpoint_finish` 两个函数的处理（关于两个函数的关系，请参阅第 5 章相关章节）。

InnoDB 启动之前，肯定是处于 shutdown 状态的，而导致 shutdown 的原因只有两种可能性，即正常关闭及 Crash 关闭。这里所说的数据恢复，主要处理的就是针对异常关闭时的情况。当然了，有一个叫 `innodb_fast_shutdown` 的参数，如果设置为 2，也相当于是一次 Crash 了，道理也是一样的。

那可能有人就要问了，如果正常关闭（`innodb_fast_shutdown` 设置为 0 或者 1），那是不是就不执行数据库恢复了？其实不是这样的，不管如何关闭数据库，启动时都会做数据库恢复的操作，只不过正常关闭的情况下，不存在没有做过 checkpoint 的日志，或者说，最新的 checkpoint 已经在最新的 LSN 位置了，又或者说所有的数据页面都已经被刷成了最新的状态。说法可以有多种，但意义其实是一样的。

日志扫描

在开始准备做数据库恢复时，首先要做的就是从日志文件中找到最新的检查点信息。我们已经知道，在日志文件最开始的 4 个页面（每个页面 512 字节）中，存储的是用来管理日志文件及日志写入情况的信息，具体格式可以从前面看到。这里所关注的检查点信息是存储在第 1 号页面和第 3 号页面中的，即所谓的 `LOG_CHECKPOINT_1` 和 `LOG_CHECKPOINT_2`。在做检查点时，这两个存储位置是轮换着使用的。

基于此，想要找到最新的检查点位置，就需要从上面的两个位置中找到一个最大值，也就是在这个点之前所有的日志都是失效的，并且对应的数据页面都是完整的。而在这个位置之后的页面，有可能是完整的，也有可能需要做 REDO，这个决定于当时 Buffer Pool 的刷盘情况，如果正好有被淘汰出去的页面，那就是完整的，否则还需要通过 REDO 日志来恢复。

先来看一段对应的精简后的代码，如下。

```
UNIV_INTERN dberr_t
recv_recovery_from_checkpoint_start_func(
    lsn_t    min_flushed_lsn, /*!< in: min flushed lsn from data files */
    lsn_t    max_flushed_lsn) /*!< in: max flushed lsn from data files */
{
    /* loval variables ... */

    if (srv_force_recovery >= SRV_FORCE_NO_LOG_REDO) {
        ib_logf(IB_LOG_LEVEL_INFO,
            "The user has set SRV_FORCE_NO_LOG_REDO on, "
            "skipping log redo");
        return(DB_SUCCESS);
    }

    recv_recovery_on = TRUE;

    mutex_enter(&(log_sys->mutex));

    /* Look for the latest checkpoint from any of the log groups */

    /* 如上所述，这里的工作就是用来从两个Checkpoint的位置，找到最新的
       max_cp_group中保存的Checkpoint对应的信息，包括最新LSN信息、LSN对应的
       日志文件中位置信息等。前面已经知道，5.6版本之后的InnoDB都支持
       总空间超过4G大小的日志文件，所以这个位置信息包括了低32位值和高32
       位值。max_cp_field用来表示最新位置是LOG_CHECKPOINT_1还是LOG_CHECKPOINT_1*/
    err = recv_find_max_checkpoint(&max_cp_group, &max_cp_field);
    if (err != DB_SUCCESS) {
        mutex_exit(&(log_sys->mutex));
        return(err);
    }

    /* 根据前面找到的max_cp_field信息，把这个位置对应的检查点信息全部读取出来，
       并存储到log_sys->checkpoint_buf空间中，下面会用到这部分数据 */
    log_group_read_checkpoint_info(max_cp_group, max_cp_field);
    buf = log_sys->checkpoint_buf;

    /* 从上面的log_sys->checkpoint_buf中拿到最新的检查点对应的LSN值及checkpoint_no
       值。checkpoint_no就是在InnoDB做检查点时，给每一次分配的一个编号，顺序增长，
       值越大，表示这个检查点越是最近做的 */
```

```

checkpoint_lsn = mach_read_from_8(buf + LOG_CHECKPOINT_LSN);
checkpoint_no = mach_read_from_8(buf + LOG_CHECKPOINT_NO);

/* Read the first log file header to print a note if this is
   a recovery from a restored InnoDB Hot Backup */

/* 读出日志头的前4个页面 (一个页面512字节) */
fil_io(OS_FILE_READ | OS_FILE_LOG, true, max_cp_group->space_id, 0,
        0, 0, LOG_FILE_HDR_SIZE,
        log_hdr_buf, max_cp_group);

/* 从上面读取出的信息中, 找到存储了ib_logfile的文件管理中, 每一个块大小的位置。
   什么? 文件块大小可以改变? 是的, 在MySQL官方版本中, 块大小是不可以修改的, 都
   是512字节, 但Percona为了适应存储设备方面的科技进步, 就支持了这个功能。当然,
   支持是支持了, 但不用也没关系, 如果不用, 那么这个位置的值就是0, 就认为还是
   默认值512字节 */

/* 声明: 不过需要注意的是, 这里是为了说明一下这个特性在Percona中已经得到了支持。
   在本章“REDO LOG日志文件管理的用途”一节中, 之所以在说明日志文件格式时没有讲
   到这个值, 是因为在前面讲到的内容中, 在LOG_FILE_WAS_CREATED_BY_HOT_BACKUP之后,
   就没有其他内容了, 这个页面就是空的了。而Percona是将块大小的信息追加到
   这个信息之后, 做到了与官方MySQL的兼容 */
log_hdr_log_block_size = mach_read_from_4(log_hdr_buf +
                                           LOG_FILE_OS_FILE_LOG_BLOCK_SIZE);

if (log_hdr_log_block_size == 0) {
    /* 0 means default value */
    log_hdr_log_block_size = 512;
}

/* Percona在这里很亲切地问候你, 如果日志文件中存储的块大小和当前系统设置的值
   不一样, 也就是说这次数据库启动时修改了这个参数, 那么它会告诉你, 并且会给出
   友好的建议, 可以RECREATE日志文件, 很贴心 */
if (UNIV_UNLIKELY(log_hdr_log_block_size != srv_log_block_size)) {
    fprintf(stderr,
        "InnoDB: Error: The block size of ib_logfile (" ULINTPF
        ") is not equal to innodb_log_block_size.\n"
        "InnoDB: Error: Suggestion - Recreate log files.\n",
        log_hdr_log_block_size);
    return(DB_ERROR);
}

```

```

/* Start reading the log groups from the checkpoint lsn up. The
   variable contiguous_lsn contains an lsn up to which the log is
   known to be contiguously written to all log groups. */

/* 到此为止，用来做恢复的信息，都已经获取到了：
checkpoint_lsn: 表示的是从这个位置开始，后面的日志需要做APPLY操作 */
recv_sys->parse_start_lsn = checkpoint_lsn;
recv_sys->scanned_lsn = checkpoint_lsn;
recv_sys->scanned_checkpoint_no = 0;
recv_sys->recovered_lsn = checkpoint_lsn;
srv_start_lsn = checkpoint_lsn;

/* 因为文件读取需要对齐到块大小，所以recv_sys->scanned_lsn
   会做对齐处理，contiguous_lsn表示的就是对齐之后的值 */
contiguous_lsn = ut_uint64_align_down(recv_sys->scanned_lsn,
                                     OS_FILE_LOG_BLOCK_SIZE);

/* 目前，InnoDB只支持一个GROUP，所以这里的遍历实际上没有什么意义，
   这里的处理是最重要的，所做的工作就是从contiguous_lsn的位置开始
   扫描所有的日志数据，然后进一步做分析、恢复等操作 */
group = UT_LIST_GET_FIRST(log_sys->log_groups);
while (group) {
    recv_group_scan_log_recs(group, &contiguous_lsn, &group_scanned_lsn);
    group->scanned_lsn = group_scanned_lsn;
    group = UT_LIST_GET_NEXT(log_groups, group);
}

/* other codes ... */
/* 做完数据库恢复之后，要处理一下收尾工作。这个收尾工作非常重要，
   类似于一个工程，在工作实施完成之后，还有一步是最后验收，验收的
   时候一般会打上一个验收合格的标志，那么这里的操作也是同样的道理，
   具体的操作就是再做一次检查点，更新一下最新的检查点信息，这样之前
   处理的所有REDO日志就失效了，如果数据库再挂了，那也是重新洗牌，与
   这次就没有什么关系了 */
recv_synchronize_groups();
/* The database is now ready to start almost normal processing of user
   transactions: transaction rollbacks and the application of the log
   records in the hash table can be run in background. */

return(DB_SUCCESS);
}

```

上面的代码，其实就是我们所熟悉的函数 `recv_recovery_from_checkpoint_start_func` 的执行过程。归纳起来，其所做的操作包括以下两部分。

- 从日志文件的固定位置找到最新的检查点信息。
- 从最新的检查点位置开始扫描日志文件，做数据库恢复。

现在，主要的工作就落在了 `recv_group_scan_log_recs` 上面，这个函数所要做的工作，就是将 `checkpoint_lsn` 位置开始的日志分片处理，每一片为 2MB 大小，对应的精简之后的代码如下。

```
static void recv_group_scan_log_recs(
    log_group_t*    group,
    lsn_t*          contiguous_lsn,
    lsn_t*          group_scanned_lsn
)
{
    /* local variables ... */
    finished = FALSE;
    start_lsn = *contiguous_lsn;

    /* 等待分析完毕 */
    while (!finished) {

        /* RECV_SCAN_SIZE 大小为 4*16KB，也就是分片大小为 64KB，
           因为已经知道，InnoDB 的日志 LSN 的增长和数据量写入的增长是同步的。
           也就是说 LSN 加 1，表示日志就多写入一个字节，所以这里在 LSN 的计算中，加上
           64KB，表示的就是 2MB 的日志量 */
        end_lsn = start_lsn + RECV_SCAN_SIZE;

        /* 在下面这个函数中，会根据之前读出来的 LSN 所对应的日志文件偏移位置，
           将 2MB 内容读取出来，存到 log_sys->buf 中，以待后面分析 */
        log_group_read_log_seg(LOG_RECOVER, log_sys->buf, group, start_lsn,
                               end_lsn, FALSE);

        /* recv_scan_log_recs 中，会检查到日志已经分析完毕，那
           数据库的 REDO 就算基本完成了，上面的 while 循环停止，具体如何判断日志
           内容读取完毕，请待进一步的讲述 */

        finished = recv_scan_log_recs(
            (buf_pool_get_n_pages()
             - (recv_n_pool_free_frames * srv_buf_pool_instances))
```



```

        * UNIV_PAGE_SIZE,
        TRUE, log_sys->buf, RECV_SCAN_SIZE,
        start_lsn, contiguous_lsn, group_scanned_lsn);

    /* 下一个分片, 从上一个分片的结束位置开始 */
    start_lsn = end_lsn;
}
}

```

从上面的函数可以看到, 数据库恢复时会根据最新检查点的位置, 将日志不断分片读取, 然后进行分片处理, 这里再来分析一下 InnoDB 是如何做分片处理的。继续看精简之后的代码, 如下。

```

UNIV_INTERN
ibool
recv_scan_log_recs(
    ulint      available_memory,
    ibool      store_to_hash,
    const byte* buf,          /*!< in: buffer containing a log segment or garbage */
    ulint      len,          /*!< in: buffer length */
    lsn_t      start_lsn, /*!< in: buffer start lsn */
    lsn_t*     contiguous_lsn,
    lsn_t*     group_scanned_lsn)
{
    /* local variables ... */
    /* 通过finished来表示恢复过程是否已经做完, 如果做完则返回值为true */
    finished = FALSE;
    /* 存储了64KB的日志 */
    log_block = buf;
    scanned_lsn = start_lsn;
    more_data = FALSE;

    do {
        /* 读出当前块中存储的数据量, 一个块, 默认大小为512字节,
           如果没有扫描到最后一块, 这个大小就都是512, 因为日志都是连续存储的 */
        data_len = log_block_get_data_len(log_block);

        scanned_lsn += data_len;

        /* 如果当前块中的数据量大于0, 就会处理当前块 */
        if (scanned_lsn > recv_sys->scanned_lsn) {

```

```

/* recv_sys, 用来存储分析之后的日志。这里的工作是将从日志
文件中读取出来的原始数据去掉头 (12字节) 尾 (4字节) 数据之后,
将中间真正的日志取出来, 放到recv_sys所指的缓存空间中, 这部分数据
才是REDO恢复真正需要的数据, 而在日志文件中存储的原始日志 (包括头尾)
是为了更好更方便地管理而设置的, 所以在这里会有这么一个转换的步骤。*/

/* 如果recv_sys的缓存空间已快要超过分析缓冲区大小 (RECV_PARSING_BUF_SIZE
=2MB), 则说明当前recv_sys中缓存的日志太多, 并且这些日志还不能满足
APPLY的条件。此时说明日志存储出现了错误, 会在errlog中报出下面的信息,
表示Recovery可能失败了。为什么是RECV_PARSING_BUF_SIZE的大小呢?
因为InnoDB认为, 在写日志时, 不会有MTR所写的日志量超过这个值, 如果有,
则只能是日志存储或者解析出了问题 */
if (recv_sys->len + 4 * OS_FILE_LOG_BLOCK_SIZE >= RECV_PARSING_BUF_SIZE) {
    fprintf(stderr, "InnoDB: Error: log parsing"
        " buffer overflow."
        " Recovery may have failed!\n");
    recv_sys->found_corrupt_log = TRUE;
} else if (!recv_sys->found_corrupt_log) {
    /* 这里就是将当前块中真正的日志内容拿出来, 存储到recv_sys缓存中去 */
    more_data = recv_sys_add_to_parsing_buf(log_block, scanned_lsn);
}

/* 更新scanned_lsn, 表示已经扫描的LSN值已经到了这个位置 */
recv_sys->scanned_lsn = scanned_lsn;
recv_sys->scanned_checkpoint_no = log_block_get_checkpoint_no(log_block);
}

/* 从这里也可以印证上面所述, 如果一个日志块不足OS_FILE_LOG_BLOCK_SIZE (默认
512字节), 则说明整个REDO日志扫描已经结束, 已经扫描到了日志结尾的位置 */
if (data_len < OS_FILE_LOG_BLOCK_SIZE) {
    /* Log data for this group ends here */
    finished = TRUE;
    break;
} else {
    /* 没有结束则向前扫描OS_FILE_LOG_BLOCK_SIZE (512字节) 的偏移量 */
    log_block += OS_FILE_LOG_BLOCK_SIZE;
}
} while (log_block < buf + len && !finished);

*group_scanned_lsn = scanned_lsn;
/* 上面已经将当前块或之前块的日志放入到了recv_sys的缓冲区中了,

```

下面就会对这部分日志做一次处理,调用的核心函数为recv_parse_log_recs,这个函数所要做的工作,接下来会以代码讲解的方式详细讲述 */

```
if (more_data && !recv_sys->found_corrupt_log) {
    /* Try to parse more log records */
    recv_parse_log_recs(store_to_hash);

    /* 从这里看到,recv_parse_log_recs将日志进一步处理之后,如果占用的
       缓存空间大于available_memory,就需要APPLY了,而这个缓存空间就是用于
       恢复HASH表,这个HASH表后面会讲述。available_memory的大小,与Buffer Pool
       有关系,InnoDB会拿一部分Buffer Pool空间来做REDO日志的恢复。
       下面这个函数recv_apply_hashed_log_recs,
       也会在后面说到 */
    if (store_to_hash && mem_heap_get_size(recv_sys->heap) > available_memory) {
        recv_apply_hashed_log_recs(FALSE);
    }

    /* 在recv_parse_log_recs中,处理掉一部分日志之后,缓冲区中
       一般会有剩余的不完整的日志,这部分日志还不能被处理,需要等待读取
       更多的日志进来,拼接之后才能继续处理,那么这里就需要将剩余的
       这部分日志移到缓冲区最开始的位置,以便继续拼接更多的日志内容 */
    if (recv_sys->recovered_offset > RECV_PARSING_BUF_SIZE / 4) {
        /* Move parsing buffer data to the buffer start */
        recv_sys_justify_left_parsing_buf();
    }
}

return(finished);
}
```

从上面的代码中,可以知道,InnoDB为了更好地管理日志文件,将连续的日志内容以块为单位来存储,加上头尾信息,继续连续存储,而在使用它的时候,又将这些日志以块为单位读取进来,掐头去尾,拼接在一起,进一步做分析处理。下面就看一下recv_parse_log_recs是如何做日志分析的。

```
static ibool recv_parse_log_recs(
    ibool store_to_hash
)
{
    /* local variables ... */
    /* 一个大的循环,连续处理恢复缓冲区中的日志内容,
       直到处理完,或者剩下的不是一个完整的MTR为止 */
```

```
loop:
```

```

/* 当前日志缓冲区中，日志的开始位置 */
ptr = recv_sys->buf + recv_sys->recovered_offset;
/* 当前日志缓冲区中，日志的结束位置 */
end_ptr = recv_sys->buf + recv_sys->len;
if (ptr == end_ptr) {
    return(FALSE);
}

/* MLOG_SINGLE_REC_FLAG表示的是，当前日志所对应的MTR，只写了这一条日志，
所以这里就作为特殊情况特别处理了。一般情况下，初始化一个页面，或者创建
一个页面等，属于这种情况，在写日志的时候，会在日志头中加上这个标志 */
single_rec = (uint)*ptr & MLOG_SINGLE_REC_FLAG;
if (single_rec || *ptr == MLOG_DUMMY_RECORD) {
    /* The mtr only modified a single page, or this is a file op */
    old_lsn = recv_sys->recovered_lsn;

    /* 如注释所述: Try to parse a log record, fetching its type, space id,
    page no, and a pointer to the body of the log record */
    len = recv_parse_log_rec(ptr, end_ptr, &type, &space, &page_no, &body);

    /* 更新进度 */
    recv_sys->recovered_offset += len;
    recv_sys->recovered_lsn = new_recovered_lsn;

    if (type == MLOG_DUMMY_RECORD) {
        /* Do nothing */
    } else if (!store_to_hash) {
        /* In debug checking, update a replicate page
        according to the log record, and check that it
        becomes identical with the original page */
    } else if (type == MLOG_FILE_CREATE || type == MLOG_FILE_CREATE2
        || type == MLOG_FILE_RENAME || type == MLOG_FILE_DELETE) {
        /* In normal mysqld crash recovery we do not try to
        replay file operations */
    } else {
        /* 将分析出来的日志信息存到一个HASH表中，又是一层缓存，
        这是第三层。后面可以了解HASH表的管理方法 */
        recv_add_to_hash_table(type, space, page_no, body, ptr + len, old_lsn,
            recv_sys->recovered_lsn);
    }
}

```

```

} else {
    /* 与上面相反的是, 这里表示的是, 一个MTR,
       包括多个日志记录, 所以这里需要一个个地去分析处理 */
    total_len = 0;
    n_recs = 0;

    /* 这里很关键, 在前面介绍的日志记录类型中, 已经提到过关于
       MLOG_MULTI_REC_END类型的作用, 它用来标志一个MTR是不是结束
       了。如果找到了这么一条日志, 则说明前面的日志是完整的, 那这个MTR
       就是可以做APPLY的。而MTR, 为何被称为mini-transaction, 也正是因为
       事务所具备的特性是原子性, 要么全做, 要么全不做, 只有找到了
       这个标志, 才说明这个MTR (物理事务) 是完整的, 这部分日志才可以被
       APPLY。可能有人会问, 这个标志有没有可能找不到? 答案是有可能。
       如果真的找不到, 这个日志就不正常, 说明这个MTR后面一部分日志
       没有被完整地写入日志文件, 那这个逻辑事务必定未提交或未提交成功
       (如果提交, 则与参数innodb_flush_log_at_trx_commit有关), 这个MTR
       就被忽略了。不过可以肯定的是, 这个MTR也是本次数据库启动时, 涉及
       日志内容中的最后一个MTR了 (除非日志文件内容存储或者解析出错了) */
    for (;;) {
        len = recv_parse_log_rec(ptr, end_ptr, &type, &space, &page_no, &body);
        /* 没有完整内容了, 则返回, 不会继续处理了 */
        if (len == 0 || recv_sys->found_corrupt_log) {
            if (recv_sys->found_corrupt_log) {
                recv_report_corrupt_log(ptr, type, space, page_no);
            }
            return(FALSE);
        }
        total_len += len;
        n_recs++;
        ptr += len;
        if (type == MLOG_MULTI_REC_END) {
            /* Found the end mark for the records */
            break;
        }
    }
}

/* 能到这里, 说明上面已经找到了MTR的结束标志, 说明这个MTR是完整的, 这样
   就会重新处理这部分日志。啊? 重新处理? 是的, 将上面检查过的重新扫描一遍。
   不过这次就可以自信满满地去处理每一个日志记录了, 而不需要担心日志的
   原子性问题了 */

```

```

/* 不过,这里的代码是不是可以做一些优化? 对于每一个
MTR,都要扫描两遍? 这样感觉会对性能造成不小的影响。
至于如何优化,方法总是有的,事在人为,关键对于那些将Log文件设置得
很大,并且经常出现异常挂机的用户来说,他们有没有对性能的需求。方法总是
跟着需求走的,有了需求,问题自然可以解决。 */
/* Add all the records to the hash table */
ptr = recv_sys->buf + recv_sys->recovered_offset;
for (;;) {
    old_lsn = recv_sys->recovered_lsn;
    /* 继续分析日志记录,找到类型、表空间ID、页面号及日志内容 */
    len = recv_parse_log_rec(ptr, end_ptr, &type, &space, &page_no, &body);

    /* 更新进度 */
    recv_sys->recovered_offset += len;
    recv_sys->recovered_lsn = recv_calc_lsn_on_data_add(old_lsn, len);
    /* 又见MLOG_MULTI_REC_END,说明已经处理完了这个MTR,则需要继续处理下一个
    MTR。结束之后,做一次大循环,直接goto loop,从头再来。*/
    if (type == MLOG_MULTI_REC_END) {
        /* Found the end mark for the records */
        break;
    }

    /* 将每一个分析出来的日志记录,加入到HASH表中。如此看来,这个HASH
    表的管理,就是下一步要研究清楚的内容了。 */
    if (store_to_hash) {
        recv_add_to_hash_table(type, space, page_no, body, ptr + len,
                               old_lsn, new_recovered_lsn);
    }

    ptr += len;
}

/* 从头再来,下一个MTR */
goto loop;
}

```

从上面的代码中可以看出,InnoDB 拿到连续的日志内容之后,以一个 mini-transaction (MTR, 物理事务) 所包含的日志为单位做分析,再将一个 MTR 中所有的日志记录一个个地分开,存储到 HASH 表中,以便做 APPLY。那么下面再来看加入到 HASH 表中的操作是如何做的。

```

static
void
recv_add_to_hash_table(
/*=====*/
    byte    type,      /*!< in: log record type */
    ulint    space,     /*!< in: space id */
    ulint    page_no,   /*!< in: page number */
    byte*    body,      /*!< in: log record body */
    byte*    rec_end,   /*!< in: log record end */
    lsn_t    start_lsn, /*!< in: start lsn of the mtr */
    lsn_t    end_lsn)   /*!< in: end lsn of the mtr */
{
    recv_t*    recv;
    ulint      len;
    recv_data_t*  recv_data;
    recv_data_t** prev_field;
    recv_addr_t*  recv_addr;

    len = rec_end - body;
    /* 针对每一条日志记录，都会有一个recv_t的结构来存储它，其包括的成员从下面可以
       看到 */
    recv = static_cast<recv_t*>(mem_heap_alloc(recv_sys->heap, sizeof(recv_t)));

    /* 成员赋值 */
    recv->type = type;
    recv->len = rec_end - body;
    recv->start_lsn = start_lsn;
    recv->end_lsn = end_lsn;
    /* 这里很重要，可以看到，InnoDB是根据space和page_no获取一个recv_addr。
       如果没有recv_addr，就创建一个，被管理到recv_sys->addr_hash的HASH表中，这里
       出现了上面提到的HASH表，也就是说，这个HASH表的键值是space, page_no
       的组合值，也就是所有日志中对应的表空间页面，都会有这样一个缓存对象 */
    recv_addr = recv_get_fil_addr_struct(space, page_no);
    if (recv_addr == NULL) {
        recv_addr = static_cast<recv_addr_t*>(mem_heap_alloc(recv_sys->heap,
            sizeof(recv_addr_t)));

        recv_addr->space = space;
        recv_addr->page_no = page_no;
        recv_addr->state = RECV_NOT_PROCESSED;
        UT_LIST_INIT(recv_addr->rec_list);
        HASH_INSERT(recv_addr_t, addr_hash, recv_sys->addr_hash,

```



```

        recv_fold(space, page_no), recv_addr);
    recv_sys->n_addrs++;
}

/* 将当前日志记录, 放到与之对应的缓存对象中, 表示当前日志所要恢复的位置
   就是在space, page_no页面中 */
UT_LIST_ADD_LAST(rec_list, recv_addr->rec_list, recv);

/* 存储日志内容时, 会用到下面代码 */
prev_field = &(recv->data);

/* 如上面注释所述, 将日志记录的内容, 即日志体 (body)
   写入到日志记录recv_t结构对象的data中 */
while (rec_end > body) {
    len = rec_end - body;
    if (len > RECV_DATA_BLOCK_SIZE) {
        len = RECV_DATA_BLOCK_SIZE;
    }
    recv_data = static_cast<recv_data_t*>(mem_heap_alloc(recv_sys->heap,
        sizeof(recv_data_t) + len));
    *prev_field = recv_data;
    memcpy(recv_data + 1, body, len);
    prev_field = &(recv_data->next);
    body += len;
}

*prev_field = NULL;
}

```

上面这段代码让我们明白, InnoDB 将每一个日志记录分开之后, 存储到了以表空间 ID 及页面号为键值的 HASH 表中。也就是说, 相同的页面肯定是存储在一起的, 并且在同一个页面上的日志是以先后顺序挂在这个对应的 HASH 节点中的, 从而保证了 REDO 操作的有序性。

从这些代码段中可以看到, 缓存到 HASH 表之后, 应该是可以找合适的时机去 APPLY 了。那什么时候才是合适的时机呢? 返回去看到函数 `recv_scan_log_recs` 的最后调用了函数 `recv_apply_hashed_log_recs`, 那么这就是真正做 APPLY 的函数了。下面详细看一下它的实现。

```

UNIV_INTERN void recv_apply_hashed_log_recs(
    ibool allow_ibuf
)
{

```

```

/* local variables ... */
loop:
recv_sys->apply_log_recs = TRUE;
recv_sys->apply_batch_on = TRUE;
/* 遍历HASH表? 是的, 将HASH表中的每一个桶中的每一个页面做连续处理 */
for (i = 0; i < hash_get_n_cells(recv_sys->addr_hash); i++) {
    /* 遍历HASH表每一个桶中的多个地址 */
    for (recv_addr = static_cast<recv_addr_t*>(
        HASH_GET_FIRST(recv_sys->addr_hash, i));
        recv_addr != 0;
        recv_addr = static_cast<recv_addr_t*>(
            HASH_GET_NEXT(addr_hash, recv_addr))) {

        /* 针对每一个页面, 做这个页面上所有的REDO操作 */
        ulint space = recv_addr->space;
        ulint zip_size = fil_space_get_zip_size(space);
        ulint page_no = recv_addr->page_no;

        if (recv_addr->state == RECV_NOT_PROCESSED) {
            mutex_exit(&(recv_sys->mutex));

            if (buf_page_peek(space, page_no)) {
                buf_block_t* block;

                mtr_start(&mtr);
                block = buf_page_get(
                    space, zip_size, page_no,
                    RW_X_LATCH, &mtr);
                buf_block_dbg_add_level(
                    block, SYNC_NO_ORDER_CHECK);

                /* 恢复一个页面的数据, 使用了一个MTR来恢复APPLY recv_addr中
                存储的所有REDO记录。需要注意的是, 这个MTR只是用来获取页面时,
                给这个页面加锁使用的, 而不会涉及REDO操作, 因为REDO是不需要
                再写日志的, 所以不用担心这个MTR涉及的日志量太大的问题 */
                recv_recover_page(FALSE, block);
                mtr_commit(&mtr);
            } else {
                /* 此处的操作是, 如果上面的buf_page_peek没有在Buffer Pool中
                找到这个页面, 那么就从文件中将这个页面载入到Buffer Pool,
                并且预读32个页面以提高性能。恢复方法也是一样的 */

```

```

        recv_read_in_area(space, zip_size, page_no);
    }
    mutex_enter(&(recv_sys->mutex));
}
}

/* Wait until all the pages have been processed */
while (recv_sys->n_addr != 0) {
    mutex_exit(&(recv_sys->mutex));
    os_thread_sleep(500000);
    mutex_enter(&(recv_sys->mutex));
}

/* Wait for any currently run batch to end.
   如注释所述, 如果上面的操作做完了, 则需要保证这些日志APPLY之后
   在ibdata及ibd(s)中落地, 此时就会将Buffer Pool中全部的脏页刷一遍,
   以保证已经处理的这些日志失效。可能有人会问, 如果在恢复的过程中, 假设
   就是这里, 还没有做刷盘操作, 数据库又挂了, 那怎么办?
   其实没关系, 整个恢复过程中, 日志也没有写, 只是扫描了一遍, 并且有可能在
   Buffer Pool中已经写了很多页面, 有可能这些页面因为LRU已经刷过
   了, 但这些操作是可重入的, 也就是说, 数据库再启动, 可以重新做一次REDO
   操作, 直到成功为止 */
success = buf_flush_list(ULINT_MAX, LSN_MAX, NULL);
recv_sys->apply_log_recs = FALSE;
recv_sys->apply_batch_on = FALSE;

/* 将HASH表中缓存的所有内容清空 */
recv_sys_empty_hash();
mutex_exit(&(recv_sys->mutex));
}

```

到这里, 应该已经清楚了 REDO 数据库恢复的整个过程, 并且可以返回到函数 `recv_recovery_from_checkpoint_start_func` 中。看一下最后的说明, 做完 REDO 之后, 做一次检查点以说明这次数据库恢复已经完成。

各位同学有没有发现这里有一个细节, 那就是 InnoDB 在辛辛苦苦将所有日志分析并且根据不同页面通过 HASH 表进行存储之后, 特别要注意下面两点特征。

- 对于同一个页面的 REDO 记录, 必然是存储在同一个 HASH 桶中的。
- 对于某一个页面的所有日志记录, 是按照先后顺序来管理的。

这两个特征非常重要，因为 REDO 日志的 APPLY 与顺序有关系，LSN 小的必定要比 LSN 大的先做 APPLY，不然有可能造成数据的覆盖。但是这有一个前提就是同一个页面，不同页面之间是不存在这样的问题的。

那我们想想，是不是只需要保证同一个页面的日志顺序执行其所有的日志记录即可，而不同页面就没必要守这个规则了，答案是肯定的。

目前的 InnoDB 难道不是这样做的吗？在上面的代码中已经看到，它是用了一个两层循环，扫描了整个 HASH 表，慢慢地一条条地做 REDO 恢复。基于上面的分析，其实可以大胆想象一下，REDO 恢复可以实现并行恢复。按照桶的下标为键值分配线程，这样同一个桶必然会分到同一个线程中去做，自然就保证了同一个页面的执行顺序，而不同桶之间的页面是没有关系的，自然就可以并行恢复了。

啊？可以这样？这个想法，可能会让那些把日志文件设置得很大、又经常出现机器宕机问题的同学（上面已经提到了他们）心潮澎湃，这样性能提升得不只一点点了。

还是那句话，这个在需要把日志文件设置很大，并且经常出现宕机时，才会有明显的优化效果。有需求，就能解决，希望这个优化会出现在某个版本中，少一些浪费的时间。

到现在为止，REDO 日志的恢复就做完了。这个时候，才真正体现了这个“累赘”的价值，感谢有你！

上面所讲的，是使用 REDO 日志来恢复数据库的过程，做完之后，整个数据库就是完整的了，保证了所有的数据库表都没有丢数据的情况，所有的数据库页面也已经是完整的了。假设此时对数据库做 DML 操作，也是可以的了。但还有一个问题没有处理，那就是此时的数据库存在脏数据。因为有些事务没有提交，但数据已经存在了（举一个例子，事务在做的过程中，日志已经写完并刷盘，就是没有提交，此时数据库挂了），那根据事务的 ACID 特性，这样的数据就不应该存在，此时，InnoDB 需要做的就是把这些事务回滚掉，这就用到了下面将要讲的“数据库回滚”。

数据库回滚

在第 5 章讲述 InnoDB 存储引擎启动的那一节中，已经知道，在调用了函数 `recv_recovery_from_checkpoint_start` 之后，又调用了 `recv_recovery_from_checkpoint_finish`。这里就是用来做数据库回滚的地方，也可以看出来，InnoDB 的 REDO 是在 UNDO 之前做的，是等到物理的数据库操作都完成之后，才能在物理数据一致的基础上去做一些逻辑的操作，即 UNDO 回滚操作。

但在讲如何回滚之前，需要先了解 UNDO 日志的存储方式。

数据库 UNDO 段管理

回滚段的管理，也是有一个入口位置用来存储回滚段的管理信息。在 InnoDB 中，是用第 6 个页面（5 号）来管理的，这个页面是专门用来存储事务相关信息的，先来看一下其页面格式，如下。

```

/** Transaction system header */
/*-----@{ */
#define TRX_SYS_TRX_ID_STORE    0    /*!< the maximum trx id or trx
        number modulo
        TRX_SYS_TRX_ID_UPDATE_MARGIN
        written to a file page by any
        transaction; the assignment of
        transaction ids continues from
        this number rounded up by
        TRX_SYS_TRX_ID_UPDATE_MARGIN
        plus
        TRX_SYS_TRX_ID_UPDATE_MARGIN
        when the database is
        started */
#define TRX_SYS_FSEG_HEADER 8    /*!< segment header for the
        tablespace segment the trx
        system is created into */
#define TRX_SYS_RSEGS          (8 + FSEG_HEADER_SIZE)
        /*!< the start of the array of
        rollback segment specification
        slots */

```

上面定义的是第 6 号页面中存储的信息及其对应的位置，每一项的详细意义如下。

- TRX_SYS_TRX_ID_STORE：用来存储事务号，在每次新启动一个事务时，都会去检查当前最大事务号是不是达到了 TRX_SYS_TRX_ID_WRITE_MARGIN (256) 的倍数，如果达到了，就会将最大的事务号写入这个位置，在下次启动时，将这个值取出来，再加上一个步长 (256)，来保证事务号的唯一性，其实就是一个经典取号器的实现原理。
- TRX_SYS_FSEG_HEADER：用来存储事务段信息。
- TRX_SYS_RSEGS：这是一个数组，InnoDB 有 128 个回滚段，那这个数组的长度就是 128，每一个元素占用 8 个字节，对应的一个回滚段存储的内容包括回滚段首页面的表空间 ID 号及页面号。

而针对每一个回滚段，即上面数组中的一个元素，也有其自己的存储格式，代码中的宏定义如下。



```
#define TRX_RSEG_MAX_SIZE 0 /* Maximum allowed size for rollback
                             segment in pages */
#define TRX_RSEG_HISTORY_SIZE 4 /* Number of file pages occupied
                                by the logs in the history list */
#define TRX_RSEG_HISTORY 8 /* The update undo logs for committed
                             transactions */
#define TRX_RSEG_FSEG_HEADER (8 + FLST_BASE_NODE_SIZE)
                             /* Header for the file segment where
                             this page is placed */
#define TRX_RSEG_UNDO_SLOTS (8 + FLST_BASE_NODE_SIZE + FSEG_HEADER_SIZE)
                             /* Undo log segment slots */
```

上面这些信息的存储,是从页面偏移 38 的位置开始的,在这个位置之前,存储的是文件管理信息(请参考第 8 章),从 38 开始,存储了上面五个信息,它们的意义分别如下。

- TRX_RSEG_MAX_SIZE: 回滚段管理页面的总数量,即所有 undo 段页面之和,一般为 ULINT_MAX,即无上限。
- TRX_RSEG_HISTORY_SIZE: 这个表用来表示当前 InnoDB 里,在 History List 中有多少个页面,即需要做 PURGE 的回滚段页面的个数。
- TRX_RSEG_HISTORY: 用来存储 History List 的链表首地址,事务提交之后,其对应的回滚段如果还不能 PURGE,就都会加入到这个链表中。
- TRX_RSEG_FSEG_HEADER: 用来存储回滚段的 Inode 位置信息,通过这个地址,就可以找到这个段的详细信息。
- TRX_RSEG_UNDO_SLOTS: 这个位置所存储的是一个数组,长度为 1024,每一个元素是一个页面号,初始化为 FIL_NULL,即空页面。

这五个信息,存储了一个回滚段的信息,最后一个位置的数组,用来真正存储回滚段的位置,后面会讲到这 128*1024 个槽是如何使用的。

根据上面的讲述,现在已经知道所有回滚段的存储架构了,如图 11.12 所示。

现在就可以知道,InnoDB 中支持的回滚段总共有 $128 \times 1024 = 131072$ 个,TRX_RSEG_UNDO_SLOTS 数组的每个元素都会指向一个页面,这个页面对应一个段,页面号就是段首页的页面号。

在每一个事务开始的时候,都会分配一个 rseg,就是从长度为 128 的数组中,根据最近使用的情况,找到一个临近位置的 rseg,在这个事务的生命周期内,被分配的 rseg 就会被这个事务所使用。

在事务执行的过程中,会产生两种回滚日志,一种是 INSERT 的 UNDO 记录,一种是 UPDATE 的 UNDO 记录,可能有人会问 DELETE 哪去了?其实是包含在 UPDATE 的回滚记录中,因为

InnoDB 把 UNDO 分为两类，一类就是新增，也就是 INSERT，一类就是修改，就是 UPDATE，分类的依据就是事务提交后要不要做 PURGE 操作，因为 INSERT 是不需要 PURGE 的，只要事务提交了，那这个回滚记录就可以丢掉了，而对于更新和删除操作而言，如果事务提交了，还需要为 MVCC 服务，那就需要将这些日志放到 History List 中去，等待去做 PURGE，以及 MVCC 的多版本查询等，所以分为两类。

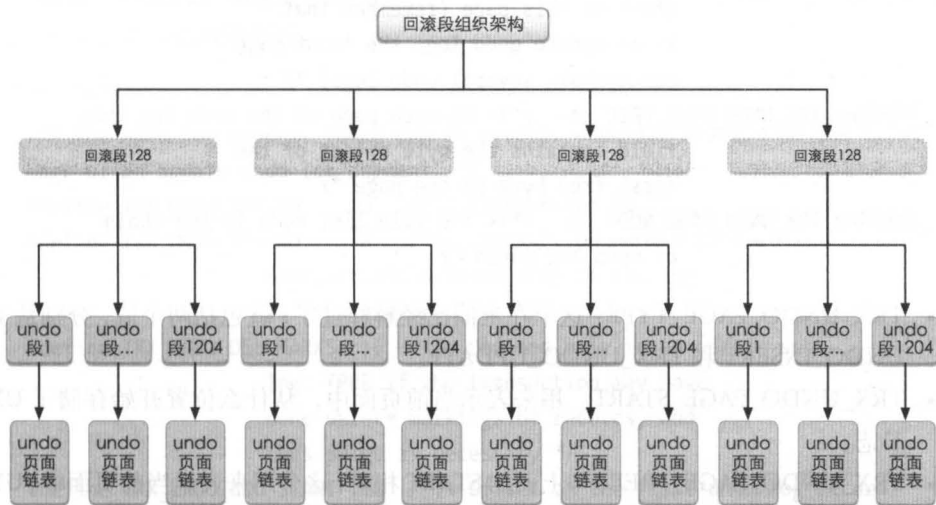


图 11.12

所以，一个事务被分配了一个 rseg 之后，通常情况下，如果一个事务中既有插入，又有更新（或删除），那么这个事务就会对应两个 UNDO 段，即在一个 rseg 的 1024 个槽中，要使用两个槽来存储这个事务的回滚段，一个是插入段，一个是更新段。


在事务要存储回滚记录的时候，事务就要从 1024 个槽中，根据相应的更新类型（插入或者更新）找到空闲的槽来作为自己的 UNDO 段。如果已经申请过相同类型的 UNDO 段，就直接使用，否则就需要新建一个段，并将段首页号写入这个 rseg 长度为 1024 的数组的对应位置（空闲位置）中去，这样就将具体的回滚段与整个架构联系起来了。

如果在 1024 个槽中找不到空闲的位置，那么这个事务就会被回滚掉，报出错误：“Too many active concurrent transactions”，错误号为 1637 的异常。当然，这种情况一般不会见到，如果能把这个用完，估计数据库已经根本动不了了。

上面讲述了整个回滚段存储架构及与事务的相关性，具体到一个事务所使用的某个回滚段的管理，就存储在了回滚段首页中，管理信息包括 3 部分，分别是 Undo page header、Undo segment header 及 Undo log header。下面来分别介绍。

1. Undo page header.

```


 /** Transaction undo log page header offsets */
#define TRX_UNDO_PAGE_TYPE 0 /*!< TRX_UNDO_INSERT or
                                TRX_UNDO_UPDATE */
#define TRX_UNDO_PAGE_START 2 /*!< Byte offset where the undo log
                                records for the LATEST transaction
                                start on this page (remember that
                                in an update undo log, the first page
                                can contain several undo logs) */
#define TRX_UNDO_PAGE_FREE 4 /*!< On each page of the undo log this
                                field contains the byte offset of the
                                first free byte on the page */
#define TRX_UNDO_PAGE_NODE 6 /*!< The file list node in the chain
                                of undo log pages */

```

- TRX_UNDO_PAGE_TYPE: 这个在上面已经解释过了, 就包括两个值, 分别是 TRX_UNDO_INSERT 和 TRX_UNDO_UPDATE。
- TRX_UNDO_PAGE_START: 用来表示当前页面中, 从什么位置开始存储了 UNDO 日志。
- TRX_UNDO_PAGE_FREE: 与上面的 START 相对, 这个用来表示当前页面中, UNDO 日志的结束位置, 也表示从这个位置开始, 可以继续追加 UNDO 日志, 直到页面存储满为止。
- TRX_UNDO_PAGE_NODE: 一个 UNDO 段中所有的页面, 通过一个双向链表来管理, 这个位置存储的就是双向链表的指针。

2. Undo segment header.

```

 /** Undo log segment header */
#define TRX_UNDO_STATE 0 /*!< TRX_UNDO_ACTIVE, ... */
#define TRX_UNDO_LAST_LOG 2 /*!< Offset of the last undo log header
                                on the segment header page, 0 if
                                none */
#define TRX_UNDO_FSEG_HEADER 4 /*!< Header for the file segment which
                                the undo log segment occupies */
#define TRX_UNDO_PAGE_LIST (4 + FSEG_HEADER_SIZE)
                                /*!< Base node for the list of pages in
                                the undo log segment; defined only on
                                the undo log segment's first page */

```

- TRX_UNDO_STATE: 用来存储当前 UNDO 段的状态, 状态包括 TRX_UNDO_ACTIVE, TRX_UNDO_CACHED、TRX_UNDO_TO_FREE、TRX_UNDO_TO_PURGE、TRX_UNDO_PREPARED 五种。

- TRX_UNDO_LAST_LOG: 用来存储最后一个 UNDO 日志的偏移位置, 用来在一个 UNDO 段中, 找到最后一个 UNDO 日志。
- TRX_UNDO_FSEG_HEADER: 这个位置, 就是用来存储当前 UNDO 段的 Inode 信息的, 通过这个信息可以知道本 UNDO 段的详细信息。
- TRX_UNDO_PAGE_LIST: 段内所有的页面都是通过链表连接起来的, 这个位置是链表的首地址, 用来管理这个链表, 上面已经介绍的 TRX_UNDO_PAGE_NODE 则是每个节点的双链指针。

3. Undo log header.



```

/** The undo log header. There can be several undo log headers on the first
page of an update undo log segment. */
#define TRX_UNDO_TRX_ID    0  /*!< Transaction id */
#define TRX_UNDO_TRX_NO    8  /*!< Transaction number of the
transaction; defined only if the log
is in a history list */
#define TRX_UNDO_DEL_MARKS 16 /*!< Defined only in an update undo
log: TRUE if the transaction may have
done delete markings of records, and
thus purge is necessary */
#define TRX_UNDO_LOG_START 18 /*!< Offset of the first undo log record
of this log on the header page; purge
may remove undo log record from the
log start, and therefore this is not
necessarily the same as this log
header end offset */
#define TRX_UNDO_XID_EXISTS 20 /*!< TRUE if undo log header includes
X/Open XA transaction identification
XID */
#define TRX_UNDO_DICT_TRANS 21 /*!< TRUE if the transaction is a table
create, index create, or drop
transaction: in recovery
the transaction cannot be rolled back
in the usual way: a 'rollback' rather
means dropping the created or dropped
table, if it still exists */
#define TRX_UNDO_TABLE_ID  22 /*!< Id of the table if the preceding
field is TRUE */
#define TRX_UNDO_NEXT_LOG  30 /*!< Offset of the next undo log header
on this page, 0 if none */
#define TRX_UNDO_PREV_LOG  32 /*!< Offset of the previous undo log

```

```

        header on this page, 0 if none */
#define TRX_UNDO_HISTORY_NODE 34 /*!< If the log is put to the history
        list, the file list node is here */

```

这是一个针对 UNDO 日志的头信息,一个事务写入一次 UNDO 日志就会创建一个 UNDO 日志单元,都会对应一个这样的 UNDO 日志头信息,用来管理这个日志信息的状态,存储一些相关的信息以备恢复时使用,多个 UNDO 日志之间,通过双向链表连接起来(通过即将介绍的 TRX_UNDO_NEXT_LOG 及 TRX_UNDO_PREV_LOG 来管理)。

- TRX_UNDO_TRX_ID: 用来存储当前 UNDO 日志对应事务的事务 ID 号。
- TRX_UNDO_TRX_NO: 事务序列号,在恢复时使用,这个序列号就是前面讲的 TRX_SYS_TRX_ID_STORE 位置存储的 ID 值。这个与上面 ID 的区别是,NO 用来在回滚时保持顺序使用,而 ID 是在事务运行时使用的。
- TRX_UNDO_DEL_MARKS: 用来表示当前 UNDO 日志中有没有通过打标志删除过记录的操作,并决定是不是要做 PURGE 操作。
- TRX_UNDO_LOG_START: 用来存储当前页面中,第一个 UNDO 日志的开始位置。
- TRX_UNDO_XID_EXISTS: 用来标志当前日志中有没有包含 Xid 事务。
- TRX_UNDO_DICT_TRANS: 用来标志当前日志对应的事务是不是 DDL 的,用来在回滚时判断如何操作。
- TRX_UNDO_TABLE_ID: 与上一个相关,如果上面的标志是真的,则这个标志的是 DDL 的表 ID。
- TRX_UNDO_NEXT_LOG: 用来链接当前 UNDO 段中所有的 UNDO 日志,这个是指向下一个 UNDO 日志。
- TRX_UNDO_PREV_LOG: 与上一个对应,这个用来指向上一个 UNDO 日志,从而构成双向链表。
- TRX_UNDO_HISTORY_NODE: 用来存储在 History List 中的双向链表指针。而这个链表的首地址,是在之前介绍的 TRX_RSEG_HISTORY 位置,可以回到前面去查看相关信息。

到目前为止,关于具体一个 UNDO 段中每个页面及页面内容是如何管理的已经讲清楚了。当一个事务需要写入 UNDO 日志时,就可以直接从对应的 UNDO 段中找到一个页面及对应的追加日志的偏移位置,然后将对应的 UNDO 日志写入即可。

数据库 UNDO 日志记录格式

在存储已经搞定之后,那么还需要继续研究一个要写入的 UNDO 日志记录的格式是什么样子的。关于记录格式,之前也介绍过 InnoDB 表中行记录 (Compact) 的格式,也介绍了 REDO 日志的记录格式,其实都是本着省空间、高效率的宗旨来设计的,那么对于 UNDO 记录也

是一样，但是因为 UNDO 日志有多个类型，针对不同的类型，其格式也不尽相同，UNDO 日志的类型有下面四种。

- TRX_UNDO_INSERT_REC: 记录插入的 UNDO 日志类型，插入记录用于回滚时，只需要通过其主键就可以实现回滚操作，所以在 UNDO 日志中，只记录了表 ID 及主键信息。回滚时，只需要通过记录中存储的主键，在原 B+ 树中找到对应的记录，然后将其删除即可。
- TRX_UNDO_UPD_EXIST_REC: 更新一条存在记录的 UNDO 日志类型。在日志内容中，需要记录的除了表 ID 信息之外，还需要记录每一个被更新的列的原始值和新值，同时还需要记录主键信息用于回滚时的检索。回滚时，还是根据主键信息，找到对应的记录，然后以旧换新，恢复原值即可。
- TRX_UNDO_UPD_DEL_REC: 更新一条已经打了删除标志记录的 UNDO 日志类型。格式与上面是一样的，回滚方法也同上。
- TRX_UNDO_DEL_MARK_REC: 删除记录时对记录打删除标志的 UNDO 日志类型，格式与上面插入操作的 UNDO 日志格式一样，只需要存储主键信息和表 ID 信息，用来在回滚或者 PURGE 时找到对应的记录即可。回滚时，根据主键信息，找到对应的记录，然后将删除标志去掉即完成回滚。

除了上面说到的 Table ID 信息、主键信息之外，还会包括一些公有的信息，比如回滚段指针、最近更新事务号，这样方便 MVCC 在回溯记录时可以找到以前的版本，关于 MVCC 的内容在这里就不详细展开了。

再回到记录格式。因为记录格式都不尽相同，所以这里只拿 TRX_UNDO_INSERT_REC 来举例说明，图 11.13 即为其格式。

end of rec 2byte	undo type 1byte	undo no Compressed int64	table id Compressed int64	trx_id Compressed int64	roll_ptr Compressed int64	n_unique*(fild_len+fild_data) n_unique*(Compressed int+datalen)	start of rec 2byte
---------------------	-----------------------	--------------------------------	---------------------------------	-------------------------------	---------------------------------	---	-----------------------

图 11.13

每一个位置的解释如下。

- 可以看到在整个记录最前面的两个字节和最后面的两个字节是用来方便找到每一个记录的，并且通过这两条信息，就可以找到每一个 UNDO 页面中的所有记录，相当于是一个由 UNDO 记录组成的双向链表，因为对于 UNDO 记录，回滚过程是一个反向操作的过程，所以需要从后向前的搜索功能。
- 第二个位置存储的就是 UNDO 记录类型。
- 第三个位置存储的是一个事务的 undo_no，用来区分一个事务中的多个 UNDO 日志的顺序。

- 接下来的位置用来存储当前回滚记录对应的表 ID，接下来的 `trx_id` 存储的就是更新这条记录时的事务 ID，即当前事务的事务 ID。
- 再接着，`roll_ptr` 用来存储当前被更新记录的上一个版本在回滚段中的位置，即这条记录中隐式列 `roll_ptr` 的值（用来在读取数据时可以找到老的版本），而当前记录的这个列的值，在写完这条 UNDO 日志之后，即将被修改为当前 UNDO 日志的位置，从而实现了一个隐式的单链表，可以使用 `roll_ptr` 的值一直回溯到第一次更新之前的版本。
- 再接下来的位置，存储的就是真实的主键信息了，存储格式是用前面若干个字节存储列数据的长度，而后面接着其数据，这样依次将所有的主键列存储完。
- 最后的位置，在这一点中已经介绍过了。

从图 11.13 中可以看到，很多位置的存储都是压缩存储的，所以上面第六点说到，列数据长度用的字节个数有可能是若干个，这决定于 InnoDB 所使用的压缩编码方式。

这里需要注意的一点是，与 REDO 日志记录存储不同，UNDO 日志的存储，是不会跨页面的，所以在页面头中关于日志存储的开始位置和结束位置就至关重要了。

其他类型的回滚记录，这里就不再介绍了，大致结构是一样的，只不过内容可能不尽相同。

需要注意的一点是，假如一个表中有多个索引，在修改一行数据时，回滚日志中也只会记录聚簇索引中的信息，而其他二级索引是不会被记录的。这是因为聚簇索引和二级索引中的每一行都是一一对应的，所以不同操作对聚簇索引操作时，也都会对二级索引有相应的操作，这样就没必要对二级索引写回滚日志了。

回滚时刻

前面已经介绍过，UNDO 日志的正确性是通过 REDO 的恢复来保证的，在 REDO 日志恢复完成之后，UNDO 操作就可以安全地进行了。数据库启动过程中，执行了用于 REDO 恢复的函数 `recv_recovery_from_checkpoint_start` 之后，就可以处理 UNDO 的数据了，InnoDB 通过函数 `trx_sys_init_at_db_start` 来将所有回滚段相关的 128×1024 个 UNDO 扫描出来（如果存在就找到，不存在就忽略），找到之后，每一个 UNDO 段的状态都已经清楚了，然后将它们都缓存起来。

然后再通过函数 `trx_lists_init_at_db_start` 依次处理每一个 UNDO 段，根据 UNDO 段的状态，决定后面将采取什么措施，如果状态为 `TRX_UNDO_PREPARED` 和 `TRX_UNDO_ACTIVE`，则这个 UNDO 段是需要做回滚操作的，否则是不需要的。决定回滚需求之后，再将最多 128×1024 个 UNDO 段按照上面提到的 `TRX_UNDO_TRX_NO` 从大到小的顺序排序。

最后就在之前介绍关于 InnoDB 存储引擎启动时的函数 `recv_recovery_from_checkpoint_finish` 中，来做回滚的相关工作。在这个函数的最后可以看到以下内容。

```

/* Roll back any recovered data dictionary transactions, so
   that the data dictionary tables will be free of any locks.
   The data dictionary latch should guarantee that there is at
   most one data dictionary transaction active at a time. */
if (srv_force_recovery < SRV_FORCE_NO_TRX_UNDO) {
    trx_rollback_or_clean_recovered(FALSE);
}

```

它根据参数 `innodb_force_recovery` 来决定要不要做回滚操作，如果设置为 3 或 3 以上，就不回滚了，这样可能导致数据库逻辑上的不一致。

最终，InnoDB 通过 `trx_rollback_or_clean_recovered` 来做回滚操作，通过扫描上面排序之后的链表，发现其还是以从大到小的顺序遍历，这个顺序很重要，因为 UNDO 是反向操作，所以应该是先处理新产生的事务，后处理老的事务，通过事务号来区分新老关系。

针对每一个 UNDO 段，InnoDB 会将所有状态为 ACTIVE 的事务的 UNDO 日志扫描出来，然后一条一条地做回滚操作，UNDO 日志记录格式已经明确，扫描所有的日志就变得非常简单，并且针对不同的操作，对应的回滚方式也已经清楚，等待所有的回滚段处理完成之后，整个数据库的回滚操作也就完成了。回滚过程如图 11.14 所示。

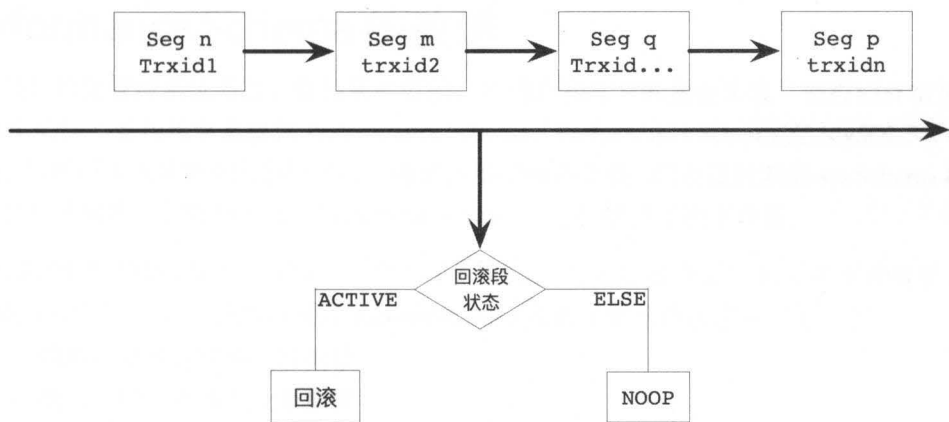


图 11.14

到这里，InnoDB 就可以继续启动了，此时的数据库处于一个完整的、可以正确提供线上服务的状态。

总结

关于日志的实现，是一个自成一套的理论体系。开头已经讲过，在几十年前就有了这方面的相关论文，并且目前基本所有大型数据库都是以这个为理论基础的，InnoDB 也不例外。

日志在数据库实现中所涉及的内容非常多，本章只能尽可能地将它们连贯起来，搞清楚它们之间的关系，以保证在日常学习及运维中，做到相互联系、辩证地看问题，以看到问题的本质。

本章提到的每一部分都可以自成一章，但这不是本书的初衷，能够让对 MySQL (InnoDB) 感兴趣的同学能将所有这些模块连贯起来，融会贯通，做到对其尽可能多地理解就够了。

MySQL 5.7 中崭新的 MySQL sys Schema

Performance Schema 的改进

对于任何数据管理系统而言，监控是必要的，从用户和客户的角度来说，监控同样很重要。MySQL 的核心监控策略是提供 Performance Schema。Performance Schema 在 MySQL 5.7 中的改进，包括引入大量新加入的监控项、降低占用空间和负载，以及通过新的 sys Schema 机制显著提升易用性。在监控方面，Performance Schema 已经提供了如下功能。

- 元数据锁 (Metadata Locking)：对于了解会话之间元数据锁的依赖关系至关重要。从 MySQL 5.7.3 开始，便可以通过 `metadata_locks` 表来了解元数据锁的相关信息。
 - 哪些会话拥有哪些元数据锁
 - 哪些会话正在等待元数据锁
 - 哪些请求由于死锁被杀掉，或者锁等待超时而被丢弃
- 进度跟踪 (Stage Tracking)：跟踪长时间操作的进度 (比如 ALTER TABLE)。从 MySQL 5.7.7 开始，Performance Schema 自动提供了语句进度信息。我们可以通过 `events_stages_current` 表来查看当前事件的进度信息。
- 事务：监控服务层和存储引擎层事务的全部方面。从 MySQL 5.7.3 开始，新增了 `events_transactions_current` 表，可以通过 `setup_consumers`、`setup_instruments` 表打开事务监控，通过该表查询到当前事务的状态。如果线上数据库遇到 undo log 大量增长、数据库性能急剧下降的情况，可以通过该表查询当前是否存在处于未提交状态的事务。如果发现的确有大量事务的 STATE 处于 ACTIVE，这时可以确定数据库有大量的事务未提交。

- 内存使用：提供内存使用信息统计，有利于了解和调整服务器的内存消耗。从 MySQL 5.7.2 开始，Performance Schema 新增了内存有关的统计信息，分别从账号、访问主机、线程、用户及事件的角度统计了内存的使用情况。
- 存储程序（Stored Programs）：存储过程、存储方法、事件调度器和表触发器的检测器。在 MySQL 5.7 中的 `setup_objects` 表中，新增了 EVENT、FUNCTION、PROCEDURE、TRIGGER 的检测器。Performance Schema 用于检测该表中匹配 OBJECT_SCHEMA 和 OBJECT_NAME 的对象。

sys Schema 介绍

说到诊断 MySQL 的性能问题，都知道从 `performance_schema` 去获取想要的信息，但是其复杂程度让使用人员使用起来很不方便。在 MySQL 5.7 中，`performance_schema` 已经有 80 多张表，每张表都是各种统计信息的罗列。另外这些表和 `information_schema` 中的部分表也有关联，导致使用人员使用起来非常不便。

在 MySQL 5.7 中新增了 `sys Schema`。MySQL `sys Schema` 是一个由一系列对象（视图、存储过程、存储方法、表和触发器）组成的 database schema，它本身不采集和存储什么信息，而是将 `performance_schema` 和 `information_schema` 中的数据以更容易理解的方式总结出来归纳为“视图”。DBA 和开发人员可以通过 `sys Schema` 方便、快速地读取 Performance Schema 收集的数据。接下来看一下 `sys Schema` 的视图中的数据是从哪里来的，如图 12.1 所示，举例说明如下。

通过视图定义可以看出，数据主要来源于 `information_schema` 中的 `COLUMNS` 和 `TABLES` 表。利用 JOIN 的方式连接查询，然后进行处理、过滤等，来展示实例中的自增量情况。

`sys Schema` 可用于典型的调优和诊断用例，这些对象包括如下三个。

- 将性能模式数据汇总到更易于理解的视图。
- 诸如性能模式配置和生成诊断报告等操作的存储过程。
- 用于查询性能模式配置并提供格式化服务的存储函数。

MySQL `sys Schema` 默认包含在 MySQL 5.7 中，并提供摘要视图以回答诸如下面所列的常见问题。

- 谁占了数据库服务的所有资源？
- 哪些主机对数据库服务器的访问量最大？
- 实例上的内存都去哪里了？

```
mysql> show create table schema_auto_increment_columns\G
***** 1. row *****
View: schema_auto_increment_columns
Create View: CREATE ALGORITHM=MERGE DEFINER='mysql.sys'@'localhost' SQL SECURITY INVOKER VIEW `sch
ema_auto_increment_columns` AS select `information_schema`.`COLUMNS`.`TABLE_SCHEMA` AS `table_schema`,`info
rmation_schema`.`COLUMNS`.`TABLE_NAME` AS `table_name`,`information_schema`.`COLUMNS`.`COLUMN_NAME` AS `col
umn_name`,`information_schema`.`COLUMNS`.`DATA_TYPE` AS `data_type`,`information_schema`.`COLUMNS`.`COLUMN_
TYPE` AS `column_type`,`locate('unsigned',`information_schema`.`COLUMNS`.`COLUMN_TYPE`) = 0) AS `is_signed`
`,`locate('unsigned',`information_schema`.`COLUMNS`.`COLUMN_TYPE`) > 0) AS `is_unsigned`,`((case `information
_schema`.`COLUMNS`.`DATA_TYPE` when 'tinyint' then 255 when 'smallint' then 65535 when 'mediumint' then 167
77215 when 'int' then 4294967295 when 'bigint' then 18446744073709551615 end) >> if((locate('unsigned',`inf
ormation_schema`.`COLUMNS`.`COLUMN_TYPE`) > 0),0,1)) AS `max_value`,`information_schema`.`TABLES`.`AUTO_INC
REMENT` AS `auto_increment`,`information_schema`.`TABLES`.`AUTO_INCREMENT` / ((case `information_schema`.`
COLUMNS`.`DATA_TYPE` when 'tinyint' then 255 when 'smallint' then 65535 when 'mediumint' then 16777215 when
'int' then 4294967295 when 'bigint' then 18446744073709551615 end) >> if((locate('unsigned',`information_s
chema`.`COLUMNS`.`COLUMN_TYPE`) > 0),0,1))) AS `auto_increment_ratio` from (`INFORMATION_SCHEMA`.`COLUMNS`
join `INFORMATION_SCHEMA`.`TABLES` on(((`information_schema`.`COLUMNS`.`TABLE_SCHEMA` = `information_schema
`.`TABLES`.`TABLE_SCHEMA`) and (`information_schema`.`COLUMNS`.`TABLE_NAME` = `information_schema`.`TABLES`
`.`TABLE_NAME`)))) where ((`information_schema`.`COLUMNS`.`TABLE_SCHEMA` not in ('mysql','sys','INFORMATION_
SCHEMA','performance_schema')) and (`information_schema`.`TABLES`.`TABLE_TYPE` = 'BASE TABLE') and (`informa
tion_schema`.`COLUMNS`.`EXTRA` = 'auto_increment')) order by (`information_schema`.`TABLES`.`AUTO_INCREMENT`
/ ((case `information_schema`.`COLUMNS`.`DATA_TYPE` when 'tinyint' then 255 when 'smallint' then 65535 w
hen 'mediumint' then 16777215 when 'int' then 4294967295 when 'bigint' then 18446744073709551615 end) >> if
((locate('unsigned',`information_schema`.`COLUMNS`.`COLUMN_TYPE`) > 0),0,1))) desc,((case `information_sche
ma`.`COLUMNS`.`DATA_TYPE` when 'tinyint' then 255 when 'smallint' then 65535 when 'mediumint' then 16777215
when 'int' then 4294967295 when 'bigint' then 18446744073709551615 end) >> if((locate('unsigned',`informat
ion_schema`.`COLUMNS`.`COLUMN_TYPE`) > 0),0,1))
character_set_client: utf8
collation_connection: utf8_general_ci
1 row in set (0.00 sec)

mysql>
```

图 12.1

sys Schema 视图摘要

sys Schema 中包含了很多以各种方式总结 Performance Schema 表的视图。这些视图大多数都是成对出现，使得每组视图中的一个成员具有与另一成员相同的名称，加上一个 x\$ 前缀。例如，host_summary_by_file_io 视图汇总按主机分组的文件 I/O 及延迟。没有 x\$ 前缀的视图提供了更加友好且容易阅读的数据，x\$ 前缀的视图提供了原始数据，更多用于需要对数据进行处理的其他工具。

视图按照展示信息可以分为如下几类。

- 主机相关信息：以 host_summary 开头的视图，主要汇总了 IO 延迟的信息，从主机、文件事件类型、语句类型等角度展示文件 IO 的信息。
- innodb 相关信息：以 innodb 开头的视图，汇总了 innodb buffer page 信息和事务等待 InnoDB 锁信息。
- io 使用情况：以 io 开头的视图，总结了 io 使用者的信息，包括等待 I/O 的情况、I/O 使用量情况，从各个角度分组展示。
- 内存使用情况：以 memory 开头的视图，从主机、线程、用户、事件角度展示内存使用情况。

- 连接与会话信息: 其中, processlist 和 session 相关的视图, 总结了会话相关信息。
- 表相关信息: 以 schema_table 开头的视图, 从全表扫描、InnoDB 缓冲池等方面展示了表统计信息。
- 索引信息: 其中包含 index 的视图, 统计了索引使用情况, 以及重复索引和未使用的索引情况。
- 语句相关信息: 以 statement 开头的视图, 统计的规范化后的语句使用情况, 包括错误数、警告数、执行全表扫描的、使用临时表、执行排序等信息。
- 用户的相关信息: 以 user 开头的视图, 统计了用户使用的文件 IO、执行的语句统计信息等。
- 等待事件相关信息: 以 wait 开头的视图, 从主机和事件角度展示等待类事件的延迟情况。

sys Schema 重点视图与应用场景

上述已经简单介绍过了 sys Schema 的视图的作用, 那么, DBA 在日常的运维过程中, 哪些场景有可能会需要 sys Schema 来帮忙呢? 以及如何使用呢? 下面简单介绍几个应用场景以及对应的 sys Schema 使用。

查看表的访问量

在一般的运维中, DBA 维护了大量的数据库。每个业务上线某些 SQL 也许不会通知 DBA, 突然间某个实例的 QPS 上升, DBA 如何查看问题并快速定位到底是哪个业务引起的 QPS 上升, 或者说业务方上线一个业务, 需要评估涉及的表访问量的增长情况, 这时该怎么办? 在 sys Schema 中的 schema_table_statistics 视图, 可以帮助我们定位到表的访问量情况。

```
mysql> SELECT table_schema, table_name, sum(io_read_requests+io_write_requests) FROM
`schema_table_statistics`;
```

```
+-----+-----+-----+
| table_schema | table_name | io_read_requests+io_write_requests |
+-----+-----+-----+
| test2       | t1         | 116 |
+-----+-----+-----+
```

```
mysql> SELECT table_schema, table_name, io_read_requests+io_write_requests as
io_total FROM `schema_table_statistics`;
```

```
+-----+-----+-----+
| table_schema | table_name | io_total |
+-----+-----+-----+
```

test2	t1	97
sys	sys_config	21
+-----+	+-----+	+-----+

我们可以监控每张表访问量的变化情况，或者监控某个库的访问量变化等。如果某个库、某个表发生变化，DBA 能够及时知道每个表的访问情况。

冗余索引与未使用的索引检查

线上使用数据库实例越来越多，每个表中都会创建索引，导致每个实例的索引非常多。一般都会有索引使用率很低或者是冗余索引的情况，这些索引是完全没有必要建立的。它们不仅消耗磁盘空间，而且还影响数据库的性能，DBA 需要实时关注是否有该类索引的出现，出现时需要立即处理。那么时，sys Schema 中的 schema_redundant_indexes 和 schema_unused_indexes 可以帮助我们快速查看索引情况。

```
mysql> select * from sys.schema_redundant_indexes \G
***** 1. row *****
      table_schema: test2
      table_name: t1
      redundant_index_name: idx_user_en
      redundant_index_columns: user_en
      redundant_index_non_unique: 1
      dominant_index_name: idx_user_en_cn
      dominant_index_columns: user_en, user_cn
      dominant_index_non_unique: 1
      subpart_exists: 0
      sql_drop_index: ALTER TABLE `test2`.`t1` DROP INDEX `idx_user_en`
```

```
mysql> select * from sys.schema_unused_indexes;
+-----+-----+-----+
| object_schema | object_name | index_name |
+-----+-----+-----+
| test2        | t1          | idx_user_cn |
+-----+-----+-----+
```

针对冗余索引，DBA 应该及时清理掉。针对长期未使用的索引，DBA 应该与使用方沟通是否以后有使用该索引的 SQL 语句等情况，如果没有或暂时不使用的，可以删除掉该索引，减少磁盘压力，提高数据库性能。

表自增 ID 监控

随着 DBA 维护的数据库实例越来越多, 表信息也越来越多, 可能某张表自增量快要超过阈值了, 继而导致业务出现问题。这时需要 DBA 清楚地知道每个表的增量列的情况。那么这时候如何查询呢? 以前都是分别去查询每个表信息, 或者通过 INFORMATION SCHEMA 来获取信息。从 MySQL 5.7 以后可以用 sys Schema 中的 schema_auto_increment_columns 视图, 就能很简单地查到每个表的自增量使用情况, 甚至可以精确到某个表的自增量情况。

```
mysql> SELECT * FROM `schema_auto_increment_columns` \G
***** 1. row *****
      table_schema: test2
      table_name: t1
      column_name: id
      data_type: int
      column_type: int(11)
      is_signed: 1
      is_unsigned: 0
      max_value: 2147483647
      auto_increment: 2
      auto_increment_ratio: 0.0000
```

在该视图中, 详细地展示了表的自增量列名、数据类型、当前使用量、最大值及使用率情况。极大地方便了 DBA 快速了解数据库自增量的使用情况。甚至可以监控该使用率, 如果超过某个阈值, 可以通过告警的方式自动化的告知 DBA 某张表的自增量可能要出现问题了, 达到预警的作用, DBA 可以快速处理这些问题。

监控全表扫描的 SQL 语句

线上数据库每天跑的 SQL 语句会有很多, 有部分 SQL 会由于未使用索引而导致全表扫描, 这些 SQL 中的很大一部分会导致数据库性能急剧下降, 甚至会导致数据库并发上升, 从而使数据库响应变慢, 直到夯住。这对 DBA 来说是非常可怕的事情。DBA 需要尽早发现这些 SQL, 关注其是否可以优化。那么在 sys Schema 的 statements_with_full_table_scans 视图中, 也许能够帮助我们定位哪些 SQL 语句走了全表扫描。

```
mysql> SELECT * FROM sys.statements_with_full_table_scans where db = 'test2' \G
***** 1. row *****
      query: SELECT * FROM `t1`
      db: test2
      exec_count: 10
      total_latency: 3.85 ms
```

```

no_index_used_count: 9
no_good_index_used_count: 0
no_index_used_pct: 90
rows_sent: 60
rows_examined: 60
rows_sent_avg: 6
rows_examined_avg: 6
first_seen: 2016-12-27 18:05:19
last_seen: 2016-12-27 18:20:06
digest: 051ceaaa78a310e4bdaada9316058f0b

```

从中可以看出该语句总共执行了 10 次，其中有 9 次未使用索引。当然，我们还可以通过总延迟来计算每次消耗的时间，可以针对那些每次消耗时间比较长的全表扫描的语句，进行优化。避免对线上数据库造成性能影响。

查看实例消耗的磁盘 I/O

数据库造成磁盘 IO 的消耗，对我们来说是需要关心的。业务方经常抱怨数据库慢了，这时 DBA 需要关心数据库到底慢在哪里？如果这时磁盘 IO 消耗过大，那么 DBA 需要知道在哪些数据库文件上消耗了大量的磁盘 IO。如果 DBA 能够快速知道具体的文件消耗磁盘 IO 量，排查问题时就会简单很多。这时，sys Schema 中的 io_global_by_file_by_bytes 视图也许可以帮助我们定位一下问题。

```

mysql> SELECT file, avg_read+avg_write as avg_io FROM `io_global_by_file_by_bytes`
order by avg_io desc limit 10;

```

file	avg_io
@@datadir/sys/schema_tables_with_full_table_scans.frm	1023
@@datadir/sys/x@0024schema_tables_with_full_table_scans.frm	994
@@datadir/mysql/tables_priv.MYD	947
@@datadir/sys/x@0024ps_digest_95th_percentile_by_avg_us.frm	906
@@datadir/performance_schema/table_lock_waits_summary_by_table.frm	897
@@datadir/mysql/proxies_priv.MYD	837
@@datadir/mysql/user.MYD	815
@@datadir/sys/sys_config.TRG	720
@@datadir/mysql/innodb_index_stats.frm	676
@@datadir/performance_schema/replication_connection_status.frm	668

DBA 可以通过该查询来大致地了解磁盘 IO 消耗在哪里，哪些文件消耗的最多。DBA 可以根据该信息，针对某些表、某些库进行针对性的优化，提高数据库性能。

使用风险

性能影响

前面简单描述了 sys Schema 的情况, 并且简单描述了几个业务场景如何使用 sys Schema。但是, sys Schema 的数据来源是 information_schema 和 performance_schema, 我们都知道 performance_schema 在引入到 MySQL 中时性能消耗是巨大的。

那么, performance_schema 开启与否, 到底对性能有多大的影响? 对于这个问题, 我们使用 Sysbench 工具测试了一下, MySQL 的版本是 5.7.14。通过测试结果可以发现, 性能损耗还是比较大的, 大概在 10% 左右。测试相关结果如图 12.2 所示 (环境不同, 参数不同, 测试结果不同, 仅供参考)。

Threads	performance_schema = OFF (TPS)	performance_schema = ON (TPS)	下降的百分比
1	1593.33	1474.24	-7.4%
4	6229.31	5465.75	-12.2
8	10547.67	9355.68	-11.3%
16	15230.79	13033.79	-14.4%
32	17434.53	14626.91	-16.1%
64	17272.13	15009.05	-13.1%
128	16290.60	14691.40	-9.8%
256	14353.48	13730.89	-4.3%

图 12.2

(Sysbench Read Write Mode)

操作风险

另外, 建议尽量不要在线上大量部署通过查询 sys 或 performance_schema, 抑或是 information_schema 中的表或视图来完成一些监控、巡检等工作, 因为查询这些信息时, MySQL 会消耗大量的资源去收集相关信息, 严重的可能会导致业务请求被阻塞, 从而引起故障。所以在使用时, 请务必了解清楚, 谨慎操作。

总结

MySQL 从 5.7.7 版本开始提供了 sys Schema, 它包含了很多表和视图。DBA 可以利用它来分析很多问题。本章讲述了部分视图与使用场景, 如果想要了解更多关于这方面的信息, 可以通过官网来了解更多、更新的信息。

更多信息详见: <http://dev.mysql.com/doc/refman/5.7/en/sys-schema.html>。

13

方便的 MySQL GTID

GTID 是全局事务标识符，是 MySQL 5.6 版本开始在主从复制方面推出的重量级特性。有了 GTID，一个事务在集群中就不再孤独，在每一个节点中，都存在具有相同标识符的兄弟们和它做伴，同一个事务，在同一个节点中出现多次的情况，也不会再重现了。GTID 的出现，直接的效果就是，每一个事务在集群中具有了唯一性的意义，这在运维方面意义非凡，给 DBA 带来了很大的便利性，因为再也不需要为不断地找点而烦恼了。

从直观上可以想到 GTID 的便利性，如下。

- 根据 GTID 可以快速地知道事务最初是在哪个实例上提交的。
- 基于 GTID 搭建主从复制更加简单，确保每个事务只会被执行一次。
- 基于 GTID 复制，可以更方便地实现 Replication 的 Failover。因为不用像传统模式复制那样去找 `master_log_file` 和 `master_log_pos`。
- MySQL Group Replication 的节点间复制完全依赖 GTID。并且，在 Group Replication 集群节点进行 Recovery 重新加入到集群中的操作中，会选择一个节点作为 Donor，然后基于 Purged 的 GTID 开始同步数据。
- 同样是在 MySQL Group Replication 中，集群使用 GTID 来标记事务，或者叫冲突验证，用于跟踪每个实例上提交的事务，确定哪些事务可能有冲突。
- GTID 的引入，让每一个事务在集群事务的海洋中有了秩序，使得 DBA 在运维中做集群变迁时更加方便，能够做到心中有数。

关于 GTID 的概念和用法, 在 MySQL 官方文档的 Replication 部分里有非常详细的讲述, 这里会介绍在日常比较重要和常用的知识, 同时分享在日常工作中碰到的一点问题。

GTID 相关概念

什么是 GTID

每提交一个事务, 当前执行线程都会拿到一个唯一标识符, 此标识符不仅对其源 MySQL 实例是唯一的, 而且在给定的复制环境中的所有 MySQL 实例中也是唯一的。所有事务与其 GTID 之间都是一一对应的, GTID 的格式如下。

```
GTID = source_id:sequence_id
```

GTID 由两部分组成, source_id 和 sequence_id。source_id 是源服务器的唯一标识, 通常使用服务器的 server_uuid 来表示 source_id。sequence_id 是在事务提交时由系统顺序分配的一个序列号, 相同 source_id 值的事务对应的 sequence_id 在 Binlog 文件中是递增且连续有序的。

可以通过 SHOW MASTER STATUS 或 SHOW SLAVE STATUS 查看当前实例执行过的 GTID 信息, 它以集合的方式呈现。

```
mysql> show master status \G
***** 1. row *****
      File: mysql-bin.000002
      Position: 561
      Binlog_Do_DB:
      Binlog_Ignore_DB:
      Executed_Gtid_Set: b591f29c-c59e-11e6-a3ca-fa163e79be41:1-2
```

其中, 本实例的 source_id 为 b591f29c-c59e-11e6-a3ca-fa163e79be41, 总共提交了 2 个事务。

GTID 集合

GTID 集合是一组全局事务标识符, 格式如下。

```
gtid_set:
  uuid_set [, uuid_set] ...
  | ''

uuid_set:
  uuid:interval[:interval]...

uuid:
```

```
hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh
```

h:

```
[0-9|A-F]
```

interval:

```
n[-n]
```

```
(n >= 1)
```

GTID 集在 MySQL 服务器中有常见的几种使用方式，例如：gtid_executed 和 gtid_purged 变量存储的值为 GTID 集。其中，UUID 按照字母排序，数字之间以升序排列。

GTID 始终保存在主从实例中，可以通过检查二进制日志来确定事务的来源。此外，一旦在给定的 MySQL 实例中提交了事务，具有相同 GTID 的事务便会被该服务器忽略。而且，在主实例上提交的事务在从库上只可以应用一次，这有助于保持主从的一致性。

GTID 生命周期

GTID 的生命周期如下。

1. MASTER 产生 GTID。

在 MASTER 上执行一个事务，MASTER 将会产生一个 GTID 信息，并保存到 Binlog 中。

2. 发送 Binlog 信息到从库上。

将二进制日志信息发送到 SLAVE 所在的服务器上，并且存储在 Relay Log 中，SLAVE 读取 GTID 并设置其 gtid_next 的值为该 GTID 值，从而告知 SLAVE 必须使用此 GTID 记录下一个事务。

3. SLAVE 执行 GTID。

SLAVE 首先验证其是否已经在自己的二进制日志中使用过了该 GTID 号。如果未使用过，SLAVE 则写入该 GTID，应用其事务，并将事务写入二进制日志。SLAVE 首先读取和检查事务的 GTID，在提交事务之前，SLAVE 不仅要保证 SLAVE 没有应用具有该 GTID 的事务，而且还要保证没有其他会话已经读取了该 GTID 但尚未提交，即不允许多个客户端应用相同的事务。

4. SLAVE 不生成 GTID。

由于 gtid_next 不为空，SLAVE 不会尝试为该事务生成新的 GTID，而是从 gtid_next 中读取 GTID 值并写入二进制日志中，来标识一个事务的 GTID 值，之后在集群中都会始终对应这个 GTID 值，且不会发生变化，起到了“身份证”标签的作用。

GTID 的维护

gtid_executed 表

在 MySQL 5.7.5 版本及以上的版本中, mysql 库中新增了表 gtid_executed, 表结构如下所示。该表中的每一行表示一个 GTID 或 GTID 集合, 包括 source_uuid、集合开始和结束的事务 ID。



```
mysql> SHOW CREATE TABLE `gtid_executed` \G
***** 1. row *****
      Table: gtid_executed
Create Table: CREATE TABLE `gtid_executed` (
  `source_uuid` char(36) NOT NULL COMMENT 'uuid of the source where the transaction
was originally executed.',
  `interval_start` bigint(20) NOT NULL COMMENT 'First number of interval.',
  `interval_end` bigint(20) NOT NULL COMMENT 'Last number of interval.',
  PRIMARY KEY (`source_uuid`, `interval_start`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

mysql> select * from mysql.gtid_executed;
+-----+-----+-----+
| source_uuid                               | interval_start | interval_end |
+-----+-----+-----+
| b591f29c-c59e-11e6-a3ca-fa163e79be41 | 1              | 7            |
+-----+-----+-----+
```

只有当 gtid_mode 为 ON 或 ON_PERMISSIVE 时, GTID 才会保存在 mysql.gtid_executed 表中。GTID 存储在该表中, 不会考虑是否启用了二进制日志。

- 当未启用 Binlog 时, 每个事务都会记录到 gtid_executed 表中。
- 当启用 Binlog 时, 每个事务不仅会记录到 gtid_executed 表中, 而且当 Binlog Rotate 或服务关闭时, 服务器会将 GTID 信息写入新的二进制日志。如果服务器异常关闭, GTID 不会被存入 mysql.gtid_executed 表中, 那么在这种情况下, MySQL 在恢复时, 会将这些 GTID 信息添加到表中, 并写入 gtid_executed 系统变量中。



注意: RESET MASTER 操作会清空 gtid_executed 表。

gtid_executed 表压缩

随着数据库的不断更新, mysql.gtid_executed 表会存入很多 GTID 信息, 并且这些事务 ID 会构成一个序列, 如下。

```
mysql> select * from mysql.gtid_executed;
```

source_uuid	interval_start	interval_end
26244cab-c5c2-11e6-923a-fa163e0f2d65	1	1
26244cab-c5c2-11e6-923a-fa163e0f2d65	2	2
26244cab-c5c2-11e6-923a-fa163e0f2d65	3	3
26244cab-c5c2-11e6-923a-fa163e0f2d65	4	4

可以通过事务的间隔来代替原来的每个 GTID 信息, 来缩减磁盘空间的消耗。

```
mysql> select * from mysql.gtid_executed;
```

source_uuid	interval_start	interval_end
26244cab-c5c2-11e6-923a-fa163e0f2d65	1	4

当 MySQL 启用 GTID 时, 服务器会定期对 mysql.gtid_executed 表执行此类型的压缩, 可以通过设置 executed_gtid_compression_period 变量来控制压缩表之前允许的事务数, 从而控制压缩率。该变量的默认值是 1000, 表示表的压缩在每 1000 个事务之后执行, 设置为 0 表示不执行压缩。



注意: 当 Binlog 开启, 且 executed_gtid_compression_period 值未使用时, MySQL Binlog 轮换会引起 mysql.gtid_executed 表的自动压缩。

MySQL 中有一个单独的后台线程来执行 gtid_executed 表压缩的操作。线程信息如下。

```
mysql> SELECT * FROM performance_schema.threads WHERE NAME LIKE '%gtid%'\G
***** 1. row *****
      THREAD_ID: 47
      NAME: thread/sql/compress_gtid_table
      TYPE: FOREGROUND
      PROCESSLIST_ID: 3
      PROCESSLIST_USER: NULL
      PROCESSLIST_HOST: NULL
```

```
PROCESSLIST_DB: NULL
PROCESSLIST_COMMAND: Daemon
PROCESSLIST_TIME: 6360
PROCESSLIST_STATE: Suspending
PROCESSLIST_INFO: NULL
PARENT_THREAD_ID: 1
        ROLE: NULL
INSTRUMENTED: YES
        HISTORY: YES
CONNECTION_TYPE: NULL
        THREAD_OS_ID: 1753
```

该线程睡眠直到执行了 `executed_gtid_compression_period` 事务后, 唤醒该线程执行 `gtid_execute` 表的压缩, 然后继续睡眠, 如此循环。当禁用二进制日志并且将此变量设置为 0 时, 该线程永远不会被唤醒。

GTID 搭建主从

在日常运维中, GTID 带来的最方便的作用就是搭建和维护主从复制, 这也是 DBA 日常工作中最经常的操作了。GTID 的主从模式替代了 MySQL 前期版本中利用二进制日志文件的名称和日志位置的做法, 使用 GTID 使操作和维护都变得更加简洁和可靠。

搭建主从时, 需要注意的 MySQL 参数

相对传统模式搭建主从来说, GTID 搭建主从模式需要在配置文件中指定如下主要参数。

- `server_id`: 设置 MySQL 实例的 `server_id`, 每个实例的 `server_id` 不能一样。
- `gtid_mode = ON`: MySQL 实例开启 GTID 模式。
- `enforce_gtid_consistency = ON`: 使用 GTID 模式复制时, 需要开启此参数, 用来保证 GTID 的一致性。
- `log-bin`: MySQL 必须开启 Binlog。
- `log-slave-updates=1`: 决定 SLAVE 从 MASTER 接收到的更新且执行完之后, 执行的 Binlog 是否记录到 SLAVE 的 Binlog 中, 建议开启。
- `binlog_format = ROW`: 强烈建议 `binlog_format` 使用 ROW 格式, 其他格式可能造成数据不一致。相关内容会在本书中关于数据安全的章节中讲述。
- `skip-slave-start = 1`: 当 SLAVE 数据库启动的时候, SLAVE 不会自动开启复制。

开启 GTID

- 如果数据库已经启动，在 MySQL 5.7.6 之前需要重启 MySQL 数据库。步骤如下。
 - 第 1 步：关闭 MASTER 的写入，保证 SLAVE 与 MASTER 的数据是同步的。
 - 第 2 步：在 SLAVE 上配置参数 `skip-slave-start=1`，避免 SLAVE 启动后，继续使用传统的复制模式。
 - 第 3 步：修改配置，开启 GTID 模式，具体参数如上所述。
 - 第 4 步：重启所有的 MySQL 数据库。



注意：在 MySQL 5.7.6 之后可以在线调整，无须重启数据库，具体操作后面会讲述。在这个版本之前，因为主从节点必须要同时开启或同时关闭，所以导致在线升级 GTID 不可行，只能重启，如果有这样的需求，需要考虑与业务的配合。

- 如果数据库是新搭建的，只需要配置上面的参数，然后启动数据库即可。这样写入事务在 Binlog 中都将带有 GTID 信息。

搭建主从

根据 MASTER 的情况，常见的基于 GTID 模式的主从搭建有以下几种方式（复制的相关账号和权限都已经授予，并且 GTID 已经开启）。

- MASTER 是新搭建的且无数据：针对这种情况，直接利用 `CHANGE MASTER` 语句搭建。

```
mysql> CHANGE MASTER TO MASTER_HOST='***', MASTER_PORT=***, MASTER_USER='***', \
-> MASTER_PASSWORD='***', MASTER_AUTO_POSITION=1;
```

- MASTER 运行不久，所有的 Binlog 保留完整：针对这种情况，可以使用类似上面的方式搭建，直接利用 `CHANGE MASTER` 语句，从 MASTER 上获取所有的 GTID，然后在 SLAVE 上执行。优点是简单快捷，缺点是如果 Binlog 相对较多，SLAVE 同步时间相对较长，可能导致网络压力过大。
- MASTER 具有大量数据：针对这种情况，可能不能使用上面的第二种方法，因为最原始的 Binlog 可能已经被删除了，无法从头开始获取所有的 GTID 信息。那么需要从 MASTER 上获取数据及该数据的 GTID 范围，然后通过 SLAVE 上设置选项 (`gtid_purged`) 的方式来跳过这些 GTID。最后通过 `CHANGE MASTER` 的方式搭建主从，具体操作方法如下。
 - 利用备份方式获取 MASTER 的数据及 GTID 范围，包含了 `gtid_purged='uuid:interval[-interval]`。使用 `innobackupex` 备份会将该信息保留在 `xtrabackup_binlog_info` 文件中。
 - 利用备份的数据，搭建 SLAVE 实例。
 - 启动 SLAVE 实例，并且设置 `gtid_purged` 的值，跳过这段范围。命令为：
`SET @@GLOBAL.GTID_PURGED='uuid:interval[-interval]'`

- 利用 CHANGE MASTER 语句, 配置主从复制。
- 启动 SLAVE 复制, SLAVE 会自动跳过这段 GTID 范围, 拉取最新的 GTID 信息。

使用 GTID 案例总结

如何跳过一个 GTID

在复制中, 偶尔会遇到主键冲突或从库找不到该条记录等错误。那么如何解决呢?

在传统的复制模式中, 经常通过设置 `sql_slave_skip_counter` 参数跳过一个事件, 如下。

```
mysql> SET GLOBAL sql_slave_skip_counter = 1;
```

但是在 GTID 模式中, 如果继续执行上述操作, 就会有如下错误产生。

```
ERROR 1858 (HY000): sql_slave_skip_counter can not be set when the server is running
with @@GLOBAL.GTID_MODE = ON. Instead, for each transaction that you want to skip,
generate an empty transaction with the same GTID as the transaction
```

在 GTID 模式的复制情况下, 如果 SLAVE 发生错误, 则可以通过跳过该事务的方式恢复主从复制, 如图 13.1 所示。

从图 13.1 中可以看出, 出错事务的 Binlog 文件为 `mysql-bin.000003`, `end_log_pos` 为 426, 其开始位置为 194。那么, 可以去主库上分析一下 Binlog, 看一下发生冲突的事务是哪个。

查看 MASTER 的 Binlog 信息如图 13.2 所示。

可以看出发现冲突的事务号为 `b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:8`, 当然要确定是哪一个事务发生了冲突, 还可以直接从 `show slave status;` 结果中通过比对的方式找到冲突的位置。严谨起见, 通过对 Binlog 内容分析得知冲突事务是插入了一条数据, 主键为 4。在从库中查看这条记录是否真的存在, 如下。

```
mysql> SELECT * FROM `t1` WHERE id = 4;
+-----+
| id | user_en |
+-----+
| 4 | qcj4    |
+-----+
```

发现 SLAVE 中已经存在这条记录了。这时, 可以通过跳过该事务的方式来放弃该事务在 SLAVE 上的执行, 使 SLAVE 能够正常运行。基于 GTID 模式的复制, 跳过一个事务, 需要利用一个空事务来完成。

```

connect_retry: 0
Master_Log_File: mysql-bin.000003
Read_Master_Log_Pos: 457
Relay_Log_File: relay-bin.000005
Relay_Log_Pos: 407
Relay_Master_Log_File: mysql-bin.000003
Slave_IO_Running: Yes
Slave_SQL_Running: No
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 1062
Last_Error: Could not execute Write_rows event on table t
est1.t1; Duplicate entry '4' for key 'PRIMARY', Error_code: 1062; handler er
ror HA_ERR_FOUND_DUPP_KEY; the event's master log mysql-bin.000003, end_log_
pos 426

Skip_Counter: 0
Exec_Master_Log_Pos: 194
Relay_Log_Space: 958
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: NULL
Master_SSL_Verify_Server_Cert: No
Last_IO_Errno: 0
Last_IO_Error:
Last_SQL_Errno: 1062
Last_SQL_Error: Could not execute Write_rows event on table t
est1.t1; Duplicate entry '4' for key 'PRIMARY', Error_code: 1062; handler er
ror HA_ERR_FOUND_DUPP_KEY; the event's master log mysql-bin.000003, end_log_
pos 426
Replicate_Ignore_Server_Ids:
Master_Server_Id: 9768
Master_UUID: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41
Master_Info_File: mysql.slave_master_info
SQL_Delay: 0
SQL_Remaining_Delay: NULL
Slave_SQL_Running_State:
Master_Retry_Count: 86400
Master_Bind:
Last_IO_Error_Timestamp:
Last_SQL_Error_Timestamp: 170203 09:25:25
Master_SSL_Crl:
Master_SSL_Crlpath:
Retrieved_Gtid_Set: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:1-8
Executed_Gtid_Set: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:1-7,
c5195495-dc57-11e6-af7f-fa163e79be41:1-3
Auto_Position: 1

```

图 13.1


```

9:25:25 server id 9768 end_log_pos 259 CRC32 0x5122385f GTID last_committed=0 sequence_number=1
SET @@SESSION.GTID_NEXT= 'b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:8'/*!*/;
9:25:25 server id 9768 end_log_pos 332 CRC32 0xc3da72d311 Query thread_id=268 exec_time=0 error_code=0
SET TIMESTAMP=1486085125/*!*/;
SET @@session.pseudo_thread_id=268/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1574961152/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!*/C utf8 /*!*/;
SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=45/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
BEGIN
/*!*/;
9:25:25 server id 9768 end_log_pos 381 CRC32 0x085a7d58 Table_map: test1.t1 mapped to number 108
9:25:25 server id 9768 end_log_pos 426 CRC32 0x90934787 Write_rows: table id 108 flags: STMT_END_F

BINLOG '
BdyTWBMoJgAAQAAAH0BAAAAAGwAAAAAAAEABXRlc3QxAAJ0MQACw8CUAACOH1aCA=
BdyTWB4oJgAALQAAAKoBAAAAAGwAAAAAAAEAAgAC//wEAAAABHFjajSHR5ka
'/*!*/;
## INSERT INTO 'test1'.t1
## SET
## 81=4 /* INT meta=0 nullable=0 is null=0 */
## 82='acj4' /* VARSTRING(80) meta=80 nullable=1 is null=0 */
9:25:25 server id 9768 end_log_pos 457 CRC32 0x780f3697 Xid = 3611
COMMIT/*!*/;

```

图 13.2



```

mysql> stop slave;
Query OK, 0 rows affected (0.00 sec)

mysql> set GTID_NEXT='b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:8';
Query OK, 0 rows affected (0.00 sec)

mysql> begin;commit;
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> SET GTID_NEXT='AUTOMATIC';
Query OK, 0 rows affected (0.00 sec)

mysql> start slave;
Query OK, 0 rows affected (0.00 sec)

```

查看一下 SLAVE 的状态是否已经正常，如下。

```
mysql> SHOW SLAVE STATUS \G
***** 1. ROW *****
Slave_IO_State: Waiting for master to send event
Master_Host: 127.0.0.1
Master_User: replication
Master_Port: 3307
Connect_Retry: 60
Master_Log_File: mysql-bin.000003
Read_Master_Log_Pos: 457
Relay_Log_File: relay-bin.000006
Relay_Log_Pos: 454
Relay_Master_Log_File: mysql-bin.000003
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Master_Server_Id: 9768
Master_UUID: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41
Master_Info_File: mysql.slave_master_info
SQL_Delay: 0
SQL_Remaining_Delay: NULL
Slave_SQL_Running_State: Slave has read all relay log; waiting for more
updates
Master_Retry_Count: 86400
Master_Bind:
Retrieved_Gtid_Set: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:1-8
Executed_Gtid_Set: b2a4fb9a-dc57-11e6-a8f9-fa163e79be41:1-8,
c5195495-dc57-11e6-af7f-fa163e79be41:1-3
Auto_Position: 1
```



注意：部分信息被忽略。

此时可以想象一下，如果删除数据时，因为找不到对应记录（Binlog 为 ROW 模式）而导致复制中断，该如何处理？如果是其他的错误呢，又该如何处理呢？还有为什么会出现这样的问题，是不是从库可写了？具体问题还需要具体分析，这里的主要目的是要说明如何在 GTID 复制模式下跳过一个事务。

跳过事务，使 SLAVE 正常运行，这种方式虽然有时候在处理问题时，确实很方便，但也是非常危险的，因为可能会出现事务不一致的情况。所以，在跳过之前，分析一下 Binlog 并且记录下来，分析是否可以跳过。跳过之后，看一下主从数据是否一致，是否需要修复数据等。总之，需要具体问题具体分析，请谨慎使用。

利用 GTID 模式快速改变主从复制关系

在日常数据库运维中,经常需要调整复制的拓扑关系,例如原有的拓扑关系,如图 13.3所示。

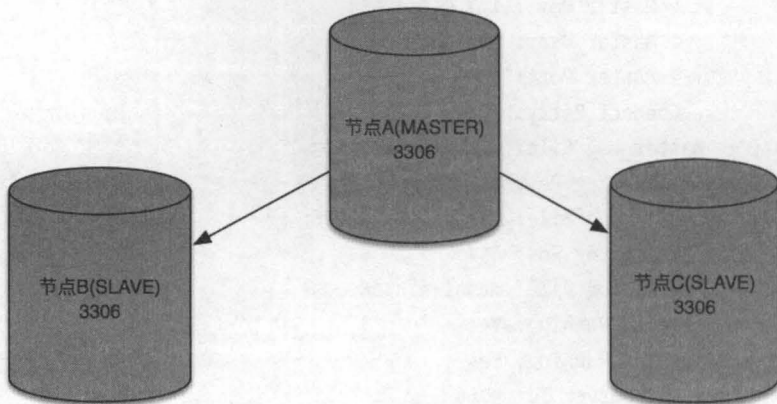


图 13.3

在维护过程中,需要将主库节点 C 的复制源修改为 B,即将复制拓扑修改为级联关系: A->B->C。具体操作如下。

1. 停止C实例的复制

```
mysql> STOP SLAVE;
```

2. 调整C实例的复制关系,修改复制源为B节点, MASTER_AUTO_POSITION设置为1

```
mysql> CHANGE MASTER TO MASTER_HOST='节点B的IP', MASTER_PORT=3306,
MASTER_AUTO_POSITION=1;
```

3. 启动C节点的复制

```
mysql> START SLAVE;
```

4. 观察复制情况

```
mysql> SHOW SLAVE STATUS \G
```

```

***** 1. row *****
Slave_IO_State: Waiting for master to send event
Master_Host: 节点B的IP
Master_User: replication
Master_Port: 3306
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Master_Server_Id: 5913

```

```

Master_UUID: a553a10a-c719-11e6-a9a9-fa163e0f2d65
Slave_SQL_Running_State: Slave has read all relay log; waiting for more
updates
Master_Retry_Count: 86400
Master_Bind:
Retrieved_Gtid_Set: a553a10a-c719-11e6-a9a9-fa163e0f2d65:1
Executed_Gtid_Set: 9cd76686-c719-11e6-a43d-fa163e0f2d65:1-2,
a553a10a-c719-11e6-a9a9-fa163e0f2d65:1

```



注意：部分信息被删除。

在 START SLAVE 操作之后，C 节点与 B 节点之间的交互过程如下。

- C 节点向 B 节点发起一个 Dump Binlog 请求，并将自身已经执行的 GTID 集合信息一起发送给 B 节点。
- B 节点通过对比接收到 C 节点发送过来的 GTID 集合，将 C 节点未执行过的 Binlog 信息发送给 C 节点。
- C 节点获取到未执行的 Binlog 信息，然后应用这些 Binlog 信息。在这个过程中，B 节点也会不断地发送最新的数据库修改信息（Binlog 信息）给 C 节点。
- C 节点不断应用这些 Binlog 信息，最终实现 C 节点与 B 节点同步。

基于 GTID 复制，DBA 可以在线快速调整复制的拓扑关系，只需要调整复制节点的基本信息，不需要手动寻找复制点，方便 DBA 在线维护。

在线将传统模式复制改为 GTID 模式复制

基于传统复制的数据库想要调整为基于 GTID 的模式复制，在老版本中是需要停服务的，这对线上环境要求比较高。但是，在 MySQL 5.7.6 版本及更新的版本中，已经支持在线修改 GTID 模式，可以在线将传统复制调整到基于 GTID 的模式复制了。具体步骤与相关说明如下。

1. 在每一台服务器上设置 ENFORCE_GTID_CONSISTENCY=WARN。

SET @@GLOBAL.ENFORCE_GTID_CONSISTENCY = WARN; 这一步设置之后，使得所有事务都允许违反 GTID 的一致性。



注意：这一步非常重要，在进行下一步之前，需要确保在错误日志中无任何警告。

2. 在每一台服务器上设置 `ENFORCE_GTID_CONSISTENCY=ON`。

`SET @@GLOBAL.ENFORCE_GTID_CONSISTENCY = ON;` 以确保所有的事务都不能违反 GTID 的一致性。

3. 在每一台服务器上设置 `GTID_MODE=OFF_PERMISSIVE`。

`SET @@GLOBAL.GTID_MODE = OFF_PERMISSIVE;` 这一步表示, 新的事务是匿名的, 同时允许复制的事务是 GTID 或是匿名的。



注意: 需要确保这一步操作在所有的服务器上都执行。

4. 在每一台服务器上设置 `GTID_MODE=ON_PERMISSIVE`。

`SET @@GLOBAL.GTID_MODE = ON_PERMISSIVE;` 这一步表示, 新的事务使用 GTID, 同时允许复制的事务为 GTID 或匿名事务。



注意: 需要确保这一步操作在所有的服务器上都执行。

5. 等待 `ONGOING_ANONYMOUS_TRANSACTION_COUNT` 状态值为 0。

`SHOW STATUS LIKE 'ONGOING_ANONYMOUS_TRANSACTION_COUNT';` 在所有从库上查询该状态, 必须要为 0 才能进行下一步。该状态表示已标记为匿名的正在进行的事务数量, 如果状态值为 0 表示无事务等待被处理。

6. 在每一台服务上设置 `GTID_MODE=ON`。

`SET @@GLOBAL.GTID_MODE = ON;` 在所有服务器上, 开启 GTID。



注意: 需要在所有的实例上执行。

7. 修改 `my.cnf` 的配置, 配置修改后, 即使数据库重启, 配置也是生效的。

`enforce_gtid_consistency = 1 gtid_mode=ON` * 上述操作是开启 GTID, 但是复制还是基于 Binlog 位置的, 可以通过将选项 `MASTER_AUTO_POSITION` 设置为 1, 将复制调整为基于 GTID 模式的复制, 具体操作如下。

```
mysql> STOP SLAVE;
mysql> CHANGE MASTER TO MASTER_AUTO_POSITION=1;
mysql> START SLAVE;
```

在线将 GTID 模式复制改为传统模式复制

上文已经描述了如何在线开启 GTID, 并且将传统模式复制改为 GTID 模式复制。接下来简单介绍如何在 GTID 模式复制下, 改为传统模式复制, 并且关闭 GTID, 而不用停止服务, 且

不影响线上业务等。具体操作步骤与说明如下。

1. 关闭基于 GTID 模式的复制，调整为传统复制。

```
mysql> STOP SLAVE; 假设复制关闭后，复制的位置为 mysql-bin.000006:518。
```

通过将选项 MASTER_AUTO_POSITION 设置为 0，调整为基于 Binlog 位置的复制。

```
mysql> CHANGE MASTER TO MASTER_AUTO_POSITION=0,  
    MASTER_LOG_FILE='mysql-bin.000006', MASTER_LOG_POS=518;
```

```
mysql> START SLAVE;
```

2. 在每一台服务器上设置 GTID 模式为 ON_PERMISSIVE。

```
SET @@GLOBAL.GTID_MODE = ON_PERMISSIVE;
```

3. 在每一台服务器上设置 GTID 模式为 OFF_PERMISSIVE。

```
SET @@GLOBAL.GTID_MODE = OFF_PERMISSIVE;
```

4. 等待所有的服务器上的变量 @@GLOBAL.GTID_OWNED 为空，它表示正在由线程执行的全局 GTID 集合。

5. 等待所有的 SLAVE 上都复制完成匿名事务。

6. 在每一台服务器关闭 GTID。

```
SET @@GLOBAL.GTID_MODE = OFF;
```

7. 修改配置文件，配置完成以后，即使数据库重启，配置也是生效的。

```
gtid_mode=OFF  
enforce_gtid_consistency=OFF
```

GTID 的限制

由于基于 GTID 的复制依赖于事务，所以在使用 GTID 时，有些 MySQL 特性不支持，如下。

- 事务中混合多个存储引擎，会产生多个 GTID。

当使用 GTID 时，如果在同一个事务中，更新包括了非事务引擎（如 MyISAM）和事务引擎（如 InnoDB）表的操作，就会导致多个 GTID 分配给同一个事务。

- 主从库的表存储引擎不一致，会导致数据不一致。

如果主从库的存储引擎不一致，例如一个是事务存储引擎，一个是非事务存储引擎，则会导致事务和 GTID 之间一对一的关系被破坏，结果导致基于 GTID 的复制不能正确地运行。

- 基于 GTID 模式复制，不支持 CREATE TABLE...SELECT 语句。

因为使用基于行模式的复制时，该语句实际上被记录为两个单独的事件，一个是创建表，

另一个是将原表中的数据插入到刚刚创建的新表中。当在事务中执行该语句时, 在一些情况下, 这两个事务可能接收到相同的事务 ID, 这意味着包含插入的事务将被从库跳过。因此, 在使用基于 GTID 的复制时, 不支持 CREATE TABLE...SELECT。



```
mysql> create table t2(id int not null auto_increment, user_en varchar(20) not
null default '', primary key(id)) as select id, name_en from t1;
ERROR 1786 (HY000): Statement violates GTID consistency: CREATE TABLE ... SELECT.
```

- 不支持 CREATE_TEMPORARY TABLE 和 DROP TEMPORARY TABLE。
使用 GTID 复制时, 不支持 CREATE TEMPORARY TABLE 和 DROP TEMPORARY TABLE。但是在 autocommit=1 的情况下可以创建临时表, MASTER 创建临时表不产生 GTID 信息, 所以不会同步到 SLAVE 上, 但是删除临时表时, 产生 GTID 会导致主从中断。
- 不推荐在 GTID 模式的实例上进行 mysql_upgrade。
因为 mysql_upgrade 的过程要创建或修改系统表 (非事务引擎), 所以不建议在开启 GTID 模式的实例上使用带有 --write-binlog 选项的 mysql_upgrade。

14

MySQL 半同步复制

MySQL 的半同步特性发布之后，一直受到业界各大公司的青睐。因为在之前完全异步的方案中，当主库挂了之后，从库很大程度上会丢失数据，导致主从数据出现不一致，并出现数据没法修复等问题。

半同步特性

MySQL 复制默认是异步的，主库写入事务在生成 Events 并写入到 Binlog 文件之后，写入线程是不等待的，或者说主库并不知道从库是不是已经收到或处理了这些 Binlog。在这种情况下，如果主库挂了，有可能没有任何一个从库可以收到已经在主库提交的事务，而此时如果高可用架构将业务从主库切换到了从库，则可能会导致从库丢失主库上面发生的很多修改。

而半同步可以作为异步同步的一个替代品，半同步复制具有以下五点特征。

- 从库会在连接到主库时告诉主库，它是不是配置了半同步。
- 如果半同步复制在主库端是开启了的，并且至少有一个半同步复制的从库节点，那么此时主库的事务线程在提交时会被阻塞并等待，结果有两种可能，要么至少一个从库节点通知它已经收到了所有这个事务的 Binlog 事件，要么一直等待直到超过配置的某一个时间点为止，而此时，半同步复制将自动关闭，转换为异步复制。
- 从库节点只有在接收到某一个事务的所有 Binlog，将其写入并 Flush 到 Relay Log 文件之后，才会通知对应主库上面的等待线程。

- 如果在等待过程中,等待时间已经超过了配置的超时时间,没有任何一个从节点通知当前事务,那么此时主库会自动转换为异步复制,当至少一个半同步从节点赶上来时,主库便会自动转换为半同步方式的复制。
- 半同步复制必须是在主库和从库两端都开启时才行,如果在主库上没打开,或者在主库上开启了而在从库上没有开启,主库都会使用异步方式复制。

通过上面几点特征,可以知道半同步的实质是,在主库被阻塞的过程中(等待从库反馈消息),主库处理线程不会返回去处理当前事务。当阻塞被激活之后,系统才会把控制权交给当前线程,然后继续处理当前事务余下的事情。处理完成之后,此时主库的事务已经提交,同时至少会有一个从库也已经收到了这个事务的 Binlog,这样就尽可能地保证了主库和从库的数据一致性。

为了弄明白半同步中的半是什么意思,这里与之前的全异步方式和全同步方式对比一下,如下。

- 对于异步复制,主库将事务 Binlog 事件写入到 Binlog 文件中,此时主库只会通知一下 Dump 线程发送这些新的 Binlog,然后主库就会继续处理提交操作,而此时不会保证这些 Binlog 传到任何一个从库节点上。
- 对于全同步复制,当主库提交事务之后,所有的从库节点必须收到、APPLY 并且提交这些事务,然后主库线程才能继续做后续操作。这里面一个很明显的缺点就是,主库完成一个事务的时间被拉长,性能降低。
- 半同步复制,是介于全同步复制与全异步复制之间的一种,主库只需要等待至少一个从库节点收到并且 Flush Binlog 到 Relay Log 文件即可,主库不需要等待所有从库给主库反馈。同时,这里只是一个收到的反馈,而不是已经完全执行并且提交的反馈,这样就节省了很多时间。

相比异步复制,半同步复制提高了数据完整性,因为很明确地知道,在一个事务提交成功之后,这个事务就至少会存在于两个地方。

半同步复制对集群整体的性能会有一些影响,因为事务提交操作由于对从库节点反馈的等待而变慢了,当然这也是对提高数据完整性的一种权衡。变慢时间的长短至少是 TCP/IP 的一次发送与接收所用的时间,因为它需要首先将 Binlog 发送出去,然后再等待从库给主库反馈消息。这也就意味着,半同步复制在网络状况好且主从主机距离较近的情况下,比网络状况不好且主从主机距离较远的情况下能够工作地更好。

上面就是半同步复制的大概原理及特性,下面就深入源码层面,对半同步的实现一探究竟,下面几节内容主要是想说明,半同步插件在开启之后,究竟是如何起作用、在哪里起作用、在每个位置的具体作用是什么。希望在看完这个表格之后,这些问题可以一清二楚。

半同步主库端

现在已经知道，在半同步环境下，主库提交任何事务，都会等待从库的反馈（ACK），而在什么时间去等待，这是一个很关键的问题。同时，半同步的实现，是通过插件的方式实现的，这个可以从安装方面看出来。那可能有些同学会问道，安装一个插件，如何能影响到 MySQL 的行为呢？这个问题就关系到了 MySQL 实现插入式的功能时所考虑的问题，简单而言，就是通过在一个事务处理过程中，在不同阶段设置“桩”的方式来实现插入式功能。在 MySQL 5.7 版本出来之后，我们现在熟悉的“桩”包括 `after_flush` 和 `after_sync`。在执行到某一个桩时，如果有对应的额外行为，就会去执行这些操作；如果没有，就会直接跳过。下面要讲的，就是以“桩”为切入点，分别讲述 MySQL 线程执行的过程中，会有哪些桩，分别是什么作用等。

下面要讲的内容中，还包括了 MySQL 源代码的一些东西，这是为了方便一些对源码感兴趣的同学找到对应的函数实现等。

`after_flush`

- 源码调用入口函数：`ordered_commit`，这也是本书所讲的关于“MySQL 5.7 多线程复制”中讲到的组提交的代码实现。
- 阶段说明：时机是在事务提交之前，`Flush Binlog` 之后，目的是取事务的结尾。因为已经做了 `Flush`，所以事务对应的 `Binlog` 结束位置就可以知道了。
- 半同步回调函数：`repl_semi_report_binlog_update`。
- 详细说明：这个桩主要是用来处理每一个事务的 `Binlog` 位置信息的。首先根据每一个提交事务对应的结束位置，保存当前实例中已经做了 `Flush` 操作的事务中最大的 `Binlog` 位置。然后，通过一个 `HASH` 表，记录下当前实例中所有提交事务的结束位置，所记录的每条信息都是由一个文件名和 `POSITION` 组成，缓存这些信息的目的是在发送 `Binlog` 时判断要不要等到从库的 `ACK`，因为只有发送一个事务的最后一个 `Binlog` 时，才需要在发送给从库的 `Binlog` 中打标志，告诉从库这个 `Event` 需要接收 `ACK`，而不是每一个发送给从库的 `Event` 都需要得到 `ACK`。

`after_sync`

- 源码调用入口函数：`ordered_commit`。
- 阶段说明：时机是在 `Flush Binlog` 之后，在 `sync` 之后，存储引擎提交之前。
- 半同步回调函数：`repl_semi_report_binlog_sync`。

- 详细说明: 这是 MySQL 5.7 版本新增的功能, 通过参数 `rpl_semi_sync_master_wait_point` (默认值为 `AFTER_SYNC`) 来控制。如果设置为 `AFTER_SYNC`, 就会执行当前这个回调。而如果设置为 `AFTER_COMMIT`, 则会执行桩 `after_commit` (下面会讲到) 对应的操作。具体这个阶段会做什么操作, 在桩 `after_commit` 的介绍中会讲, 这里只讲可能存在的问题。在 5.7 版本中, 如果参数为 `AFTER_SYNC` (默认), 假设主库在存储引擎提交之前挂了, 那么很明显这个事务是不成功的, 但由于对应的 Binlog 已经做了 Sync 操作, 从库已经收到了这些 Binlog, 并且执行成功, 相当于在从库上多了数据, 明显是存在问题的, 但多了数据, 问题一般不算严重。这个问题可以这样理解, 作为 MySQL, 在没办法解决分布式数据一致性问题的情况下, 它能保证的是不丢数据, 多了数据总比丢数据要好。但如果还是按照之前 MySQL 5.6 的设置 (相当于是 `AFTER_COMMIT`), 那么因为是在主库提交之后才去等待, 所以, 如果在等待过程中主库挂了, 但此时从库可能根本没有收到相应的 Binlog, 如果主库永远也启动不了了, 那么实际上已成功提交的事务, 在从库上是找不到的, 也就是数据丢失了, 这是 MySQL 不愿意看到的, 所以在 MySQL 5.7 版本中默认增加了 `AFTER_SYNC` 的选择并将其设置为默认值是合理的。

after_commit

- 源码调用入口函数: `finish_commit`。
- 阶段说明: 数据库存储引擎提交完成之后, 此时有可能会等待从库的消息。
- 半同步回调函数: `repl_semi_report_commit`。
- 详细说明: 这个桩是半同步中最重要的位置之一, 提交线程执行到这里时会检查当前实例中收到的从库 ACK 信息, 其中包含了 Binlog 位置信息, 这个 ACK 有可能是其他提交事务对应的 ACK。这个提交线程将自己的 Binlog 位置与最近收到的 ACK 消息中的位置对比, 如果 ACK 的更大, 就不再需要等待, 直接提交。因为一个更大的 Binlog 位置已经复制到从库中了, 当前事务对应的相对较小的 Binlog 自然也已经复制过去了, 所以不需要等待。如果还没有收到任何比当前事务对应的 Binlog 位置大或相等的 ACK, 则需要等待, 此时状态参数 `rpl_semi_sync_master_wait_sessions` 递增 1, 然后当前线程进入等待状态。等待结果有两种, 一种是超时 (超时时间为参数 `rpl_semi_sync_master_timeout` 的值), 此时就将半同步关掉; 一种是正常等到了 ACK, 当前线程就会被唤醒, 并且状态参数 `rpl_semi_sync_master_wait_sessions` 会减 1。当多个事务并发提交时, 如果有多个线程在等待 ACK, `rpl_semi_sync_master_wait_sessions` 的值可能会比较大。等到 ACK 之后, 统计出已经等待的时间, 来更新参数 `rpl_semi_sync_master_avg_trx_wait_time`、`rpl_semi_sync_master_avg_net_wait_time` 等。最后再更新参数 `rpl_semi_sync_master_yes_tx`、`rpl_semi_sync_master_no_tx` 等参数值, 表示有多少事务是通过半同步产生的、有多少不是, 可以通过这些参数来看半同步是不是真的在运作了。MySQL 5.6 版本默认是在

这个桩上等待的，而在 MySQL 5.7 版本中，如果将参数 `rpl_semi_sync_master_wait_point` 设置为 `AFTER_COMMIT`，对应的就是在桩 `after_commit` 中做这些操作了，而如果设置为 `AFTER_SYNC`，则这些操作就会在桩 `after_sync` 中做了。

after_rollback

- 源码调用入口函数：`ha_rollback_low`。
- 阶段说明：这也是一个拥有插件式特点的 MySQL Server 加的一个桩，这个桩设置在数据库回滚之后。
- 半同步回调函数：`repl_semi_report_rollback`。
- 详细说明：在 Binlog 中，经常也会出现 Rollback 操作，用来将之前产生的操作回滚掉。对于一个事务，回滚操作和提交操作的意义是相同的，即结束这个事务。对于回滚操作，这里是要在一个事务的 Binlog 中加入 Rollback 的相关处理，然后将 Binlog 复制给从库，其实这个意义和事务提交是一样的，即在这个位置，还是要等待从库的 ACK。从这个意义上讲，桩 `after_rollback` 的操作和 `after_commit` 就完全一样了。回调函数 `repl_semi_report_rollback` 直接调用了 `repl_semi_report_commit`。

transmit_start

- 源码调用入口函数：`mysql_binlog_send`。
- 阶段说明：此桩执行时机是在主库向从库发送 Binlog 之前，在从库请求 Binlog 复制之后。当主库接收到 Dump Binlog 信息准备向从库发送 Binlog 时执行此桩，每执行一次 `start slave`，也执行一次此桩。
- 半同步回调函数：`repl_semi_binlog_dump_start`。
- 详细说明：首先判断当前有没有开启半同步，如果没有，直接返回。在 `ack_receiver`（下面会详细介绍）中加入当前从库线程信息，状态参数 `Rpl_semi_sync_master_clients` 的值加 1。这里有一个假设，请求 Binlog 这个位置的从库，已经接收到了所有在这个点之前的 Binlog。所以这里的处理是，如果现在有等待从库消息的主库线程，就告诉它不要等了，此时至少会有一个主库事务可以提交了。通知方式是广播。在该假设的前提下，当前请求的位置会被主库当作是 ACK 消息。
- `ack_receiver`：`ack_receiver` 是在 MySQL 5.7 版本中新加入的 ACK 接收管理机制。这种机制是单独建立的，通过 `select` 来对每一个注册的从库做监听，在有 ACK 消息返回时，通过读取其 ACK 消息获取对应的 Binlog 位置。MySQL 5.7 通过新增参数 `rpl_semi_sync_master_wait_for_slave_count` 来设置主库，在一个事务提交之前，需要等待几个从库的 ACK，以便进一步提高半同步的安全性。如果设置为 1，则与 5.6 版本无异；而如果大于 1，则采

取 5.7 版本的新方式。在 5.7 版本中统计主库收到多少个从库的反馈是通过创建一个长度为 `rpl_semi_sync_master_wait_for_slave_count` 的数组，每个位置对应一个从库 ACK 信息（包含 Binlog 位置信息）实现的，每当一个从库给出反馈，就根据其 `server_id` 来更新对应的位置信息，当这个数组被写满时，说明所有从库都给出了反馈，同时这个数组中 Binlog 的最小位置就是这次通知主库可以提交的最大位置点（因为这是从数组中找的最小 Binlog 位置，说明数组中对应的所有从库都至少复制到了这个位置），同时将数组中与这个 Binlog 位置相同的位置清空以便循环使用，当再次都存储满之后，再从中找到新的最小 Binlog 位置，这样不断循环往复向前推进。简而言之，这类似木桶原理，木板个数为 `rpl_semi_sync_master_wait_for_slave_count`，最短的板为安全的可提交位置。MySQL 5.7 版本相比 MySQL 5.6 版本的共同点是，它们都找到了一个 Binlog 位置；而不同点是这个位置在 MySQL 5.7 中有多个从库都至少复制了这个 Binlog 位置，而在 MySQL 5.6 中只有一个节点确定复制到这个位置。在这个位置确定之后，`ack_receiver` 就会通过广播的方式告诉正在等待 ACK 消息的主库线程继续它们的提交工作。

transmit_stop

- 源码调用入口函数：`mysql_binlog_send`。
- 阶段说明：这个桩的位置是在主库向从库发送 Binlog 结束之后，结束原因不限。
- 半同步回调函数：`repl_semi_binlog_dump_end`。
- 详细说明：在主库向从库发送 Binlog 结束之后（从库断掉复制等），主库这边首先要将状态参数 `Rpl_semi_sync_master_clients` 减 1。如果当前主库的半同步还处于开启状态，同时参数 `rpl_semi_sync_master_wait_no_slave` 为 OFF、`Rpl_semi_sync_master_clients` 为 0，则此时系统会自动关闭半同步，而这也是参数 `rpl_semi_sync_master_wait_no_slave` 的意义。

before_send_event

- 源码调用入口函数：`mysql_binlog_send`。
- 阶段说明：这个桩执行位置是在发送 Binlog 事件之前，作用是在预留的位置中填写半同步控制信息。
- 半同步回调函数：`repl_semi_before_send_event`。
- 详细说明：这个桩的主要作用，就是要告诉从库在收到 Binlog 之后，要不要向主库反馈 ACK 消息，实现方法是在发送的 Binlog 事件头信息中打标志，要打什么标志，通过多个条件来判断。①首先，发送线程会确定当前要发送的 Binlog 位置是否比从库最新反馈的 Binlog 位置小。如果是的话，说明更新的 Binlog 已经收到了 ACK，而落后的 Binlog 就不需要再请求 ACK 了，这种情况一般发生在刚开始复制的时候；反过来，如果要发送

的 Binlog 位置更大,则需要进一步判断,转步骤②。②如果已经有一个对应更大的 Binlog 位置的事务在等待 ACK,则这个事务也不会再请求 ACK,否则需要进一步判断,转步骤③。③判断当前要发送的 Binlog 位置是不是一个事务的最后结束位置(在桩 after_flush 中,有说明通过一个 HASH 表来缓存事务 Binlog 结束位置的信息),如果不是,则此时发送线程不需要请求 ACK,否则说明需要 ACK 反馈消息。完成这三步之后,就可以确定在发送 Binlog 事件时告诉从库要不要反馈 ACK 消息了。最终,这个决定会通过修改当前要发送的 Binlog 事件头内容打上相应标志。如果要反馈 ACK 消息,则写入 1,否则写入 0。通过这一系列的过滤,主要目的是为了尽可能地减少主库和从库的 ACK 交互,以提高半同步的性能,让每一次 ACK 都是有用的。

after_send_event

- 源码调用入口函数: mysql_binlog_send。
- 阶段说明: 这个桩的执行时机是在 Binlog 事务发送完成之后,如果需要的话,从从库接收反馈消息,来确定从库是不是已经收到 Binlog,以便通知等待事务继续做,同时继续向从库发送 Binlog。
- 半同步回调函数: repl_semi_after_send_event。
- 详细说明: 在 MySQL 5.6 版本中,这个桩主要是用来接收从库的 ACK 的。与桩 before_send_event 对应的是,如果在 before_send_event 中确定了一个事务需要发送 ACK 请求,那么此时就会等待 ACK 反馈。如果确定需要等待 ACK 反馈(通过判断之前发送的 Binlog 事件头的标志位),则直接从当前线程与从库的连接中读取 ACK 消息,如果从库还没有发送,则当前发送线程被阻塞,直到有消息发送过来阻塞才会被解除。当前线程接收到 ACK 之后,会从中取出 Binlog 位置信息,然后更新主库中的从库,反馈 ACK 位置信息。如果 ACK 的位置比处于等待状态的主库事务线程的最小位置大,则说明可以广播通知它们可以继续提交操作了。假设某些事务之前讲的桩 after_commit 或 after_sync 处于等待状态了,则都是等待当前这个广播以继续提交。在 MySQL 5.7 版本中,接收 ACK 消息不会在这里做,它被放到一个专门的 ACK 消息接收线程 ack_receiver 中,这在桩 transmit_start 中已经介绍过了。两个版本的实现原理是一样的,只是加入了对多个从库的处理机制。

reserve_header

- 源码调用入口函数: mysql_binlog_send。
- 阶段说明: 此桩的执行时机是在每次发送一个事务的时候,目的是预留半同步通信位置。
- 半同步回调函数: repl_semi_reserve_header。

- 详细说明：前面已经说过，在 Binlog 事件头部有 4 字节的预留位置，如果当前要发送的 Binlog 对象为半同步的（通过在从库端中介绍的 `rpl_semi_sync_slave` 参数来判断），则此时预留两个字节给半同步通信使用（从库会通过这两个字节来判断是不是要给主库反馈）。

after_reset_master

- 源码调用入口函数：`reset_master`。
- 阶段说明：重置复制信息，此桩执行时机是在执行 `reset master` 语句之后。
- 半同步回调函数：`repl_semi_reset_master`。
- 详细说明：重设本地所有关于半同步的参数。

半同步从库端

thread_slave

- 源码调用入口函数：`handle_slave_io`。
- 阶段说明：从库向主库 Dump Binlog 之前，具体是在开始连接主库前。
- 半同步回调函数：`repl_semi_slave_io_start`。
- 详细说明：如果 `rpl_semi_sync_slave_enabled` 参数为 ON，此时会将 `status` 参数 `Rpl_semi_sync_slave_status` 也设置为 ON。需要注意的是，如果复制已经开始，而状态参数 `rpl_semi_sync_slave_enabled` 为 OFF，则需要将其设置为 ON 之后，还需要做 `stop slave; start slave` 才能开启半同步。也就是说，是不是启用半同步，是在这里决定的。

before_request_transmit

- 源码调用入口函数：`request_dump`。
- 阶段说明：此桩的执行时机是在上一个桩之后，具体是向主库发起 Dump Binlog 请求的时候。
- 半同步回调函数：`repl_semi_slave_request_dump`。
- 详细说明：如果从库没有开启半同步，或者主库参数 `rpl_semi_sync_master_enabled` 不存在或没有开启，则与异步复制无异。通过 `slave` 对 `master` 的连接，设置会话参数 `rpl_semi_sync_slave=1`，表示当前从库是半同步类型的复制。如果一个主有多个从，判断是不是半同步的从，就是通过这个变量来区别的，这个变量 `DBA` 是看不到的，只有系统才能知道。这个参数其实就是一个会话变量，是半同步连接自己设置的，名字是它自己起的。

after_read_event

- 源码调用入口函数：handle_slave_io。
- 阶段说明：此桩的执行时机是在连接上主库并读取到 event 之后。每读到一个事件，都会执行一次这个桩。
- 半同步回调函数：repl_semi_slave_read_event。
- 详细说明：如果参数 repl_semi_sync_slave_status 为 ON，读取 Binlog 头部信息，Binlog 头部本身有 4 个字节的预留位置，半同步使用 2 字节来同步主从行为。第 1 个字节是一个 Magic 数，用来校验半同步；第 2 个字节用来表示需不需要从库给主库发送信息的标志。

after_queue_event

- 源码调用入口函数：handle_slave_io。
- 阶段说明：此桩的执行时机是在从主库 Dump 到 Binlog 并写入本地 Relay Log 之后。每读到一个事件，都会执行一次这个桩。
- 半同步回调函数：repl_semi_slave_queue_event。
- 详细说明：如果参数 repl_semi_sync_slave_status 为 ON，同时上面的桩 after_read_event 得到的标志为 1，此时从库会给主库发送接收到的消息，即 ACK 反馈，否则就不向主库发送消息。给主库发送的消息是当前最新的 Binlog 位置，包括文件名及 POSITION。

thread_stop

- 源码调用入口函数：handle_slave_io。
- 阶段说明：此桩的执行时机是在停止复制之后。
- 半同步回调函数：repl_semi_slave_io_end。
- 详细说明：在执行 stop slave 命令时会执行这个回调函数，其实只是将参数 repl_semi_sync_slave_status 设置为 OFF 而已。

after_reset_slave

- 源码调用入口函数：reset_slave。
- 阶段说明：此桩的执行时机是在执行 reset slave 语句之后。
- 半同步回调函数：repl_semi_reset_slave。
- 详细说明：当前版本中，没有任何操作。

半同步实现

众所周知，半同步复制是一个插件，用到的时候只需要 install 即可。那么有人可能会想，一个动态加载的插件，如何能让事务处理线程等待、停下来，这是怎么做到的？

其实安装时指定的插件包，只是这个插件的实现体而已，在 MySQL 源码中还是需要支撑这个实现体的框架的。大概包括如下几个框架（框架，简而言之，就是在代码的某些位置，加入一些桩，或者叫 observer，观察者，在执行到这些桩的位置时，如果有插件在这些 observer 中注册实现体了，就执行这些实现体，否则没有任何操作）。

Trans_observer

这个 observer 是关于事务处理的，包括如下五个过程（对应表格中的“半同步回调”一列）。

- before_dml：从字面意思就可以知道，这是在执行一个 DML 之前可以做的一些操作，如果有，就实现。
- before_commit：这个过程，是在将 Binlog 写入文件之前执行的。
- before_rollback：是在存储引擎回滚时执行的。
- after_commit：是在存储引擎提交之后执行的，包括这个以及上面两个，在上面表格中都有所体现。
- after_rollback：在回滚之后执行。

Server_state_observer

这个 observer 是关于服务器状态的。当然，它与复制没有关系，这里也一并说明，包括如下六个过程。

- before_handle_connection：这个过程，在服务器已经准备好可以接受客户端的连接时调用。
- before_recovery：这个过程，在开始做数据库恢复时调用。
- after_engine_recovery：在数据库的每个存储引擎恢复完成之后调用。
- after_recovery：在执行完整个数据库的数据恢复之后调用。
- before_server_shutdown：在数据库正常关闭时调用。
- after_server_shutdown：在数据库正常关闭时调用。

Binlog_storage_observer

这个 observer 是关于 Binlog 的存储的，包括下面两个过程。

- `after_flush`: 在 Binlog 刷盘之后调用。在半同步插件中，处理的是将事务的完整结束位置写入活动事务缓存中，以便判断是否需要从库发反馈消息回来。
- `after_sync`: 这个过程是在 MySQL 5.7 版本中新增的。在半同步插件中，它是当 Binlog 在主库写入到 Binlog 文件之后，就开始等待从库的反馈消息了。

Binlog_transmit_observer

这个 observer 是关于 Binlog 的发送的，包括下面六个过程。

- `transmit_start`: 这个过程，在主库开始向从库发送 Binlog 时调用。对于一次从库的 Binlog 请求只调用一次，主要处理的是主库对从库信息的缓存及感知，并且假设请求点之前的 Binlog 都已经在从库那边了。
- `transmit_stop`: 这个过程，在主库停止向从库发送 Binlog 时调用。和上面的过程相反，主要是销毁对从库的缓存等。
- `reserve_header`: 这个过程，发生在每次向从库发送一个 Event 的时候。因为要告诉从库是否需要发送反馈给主库的空间，所以这里就要在事务头部预留出空间来（实际上是初始化头部中 4 个空闲字节中的前面 2 个字节）。
- `before_send_event`: 这个过程，发生在上预留空间之后。在发送之前，已经知道当前事件是否是一个事务的最后一个位置。如果是最后一个位置，就在预留位置设置需要反馈消息的标志，这样从库接收到之后，才会给主库发送反馈。
- `after_send_event`: 这个过程是在发送之后。可能要等待接收至少一个从库的反馈消息，因为上面发送了什么 Event，此时是知道的，如果发现预留位置上面的标志需要等待从库的反馈消息，那么此时便需要等待，直到收到反馈消息，才能继续发送 Binlog。而在 5.7 版本中，是把等待 ACK 的工作放入一个专门的接收线程中，实际上是一个原理。
- `after_reset_master`: 这个过程，是在执行 RESET MASTER 命令时调用的过程，主要是初始化一些变量状态等。

Binlog_relay_IO_observer

这个 observer 是关于从库端 Binlog 处理的，包括下面七个过程。

- `thread_start`: 这个过程，就是简单处理一下从库这边的半同步参数的。
- `thread_stop`: 这个过程，和上面的相反，也比较简单。


- `applier_stop`: 这个过程, 在半同步中没有实现, 其调用时机是在 SQL 线程执行失败, 导致复制中断的时候。
- `before_request_transmit`: 这个过程, 是从库向主库请求 Binlog 时调用的, 与上面 `Binlog_transmit_observer` 中的 `transmit_start` 对应。
- `after_read_event`: 这个过程是在每次 IO 线程读到 Binlog 中的一个事件之后, 从 Binlog 事件头部读取反馈标志来判断是否要发送反馈消息, 这个结果会在下面的过程中用到。
- `after_queue_event`: 如果上面的结果表示从库需要发送反馈消息, 那么在这个过程中, 会将当前 Binlog 的位置信息发送给主库。
- `after_reset_slave`: 这个过程, 是在执行 `RESET SLAVE` 命令时调用的。

下面是这些过程调用方式的一个例子。

```
 (void) RUN_HOOK(binlog_relay_io, after_reset_slave, (thd, mi));
```

第一个参数指定的是 observer 名; 第二个参数指定的是其中的过程名; 第三个参数指定的是具体某一个过程的参数。

上面已经提到, 介绍的这些过程或 Observer 只是一个框架, 如果要实现一个插件, 需要给这个框架注册自己的过程实现体, 针对每一个插件, 都需要有一个公共的声明结构。

```
 mysql_declare_plugin(semi_sync_master)
{
    MYSQL_REPLICATION_PLUGIN,
    &semi_sync_master_plugin,
    "rpl_semi_sync_master",
    "He Zhenxing",
    "Semi-synchronous replication master",
    PLUGIN_LICENSE_GPL,
    semi_sync_master_plugin_init, /* Plugin Init */
    semi_sync_master_plugin_deinit, /* Plugin Deinit */
    0x0100 /* 1.0 */,
    semi_sync_master_status_vars, /* status variables */
    semi_sync_master_system_vars, /* system variables */
    NULL, /* config options */
    0, /* flags */
}
mysql_declare_plugin_end;
```

其中可以看到, 有一个注释为 “Plugin Init” 对应的参数 `semi_sync_master_plugin_init`, 这是一个函数指针, 半同步插件就是通过这个函数来初始化的, 其实现代码如下。

```
static int semi_sync_master_plugin_init(void *p)
{
    if (repl_semisync.initObject())
        return 1;
    if (ack_receiver.init())
        return 1;
    if (register_trans_observer(&trans_observer, p))
        return 1;
    if (register_binlog_storage_observer(&storage_observer, p))
        return 1;
    if (register_binlog_transmit_observer(&transmit_observer, p))
        return 1;
    return 0;
}
```

可以看到, 通过函数 `register_trans_observer`、`register_binlog_storage_observer`、`register_binlog_transmit_observer`, 向框架注册了半同步插件自己实现的 `trans_observer`、`storage_observer` 及 `transmit_observer`, 它们对应的实现函数如下。

```
Trans_observer trans_observer = {
    sizeof(Trans_observer),    // len

    repl_semi_report_before_dml,    //before_dml
    repl_semi_report_before_commit, // before_commit
    repl_semi_report_before_rollback, // before_rollback
    repl_semi_report_commit, // after_commit
    repl_semi_report_rollback,    // after_rollback
};

Binlog_storage_observer storage_observer = {
    sizeof(Binlog_storage_observer), // len

    repl_semi_report_binlog_update, // report_update
    repl_semi_report_binlog_sync,   // after_sync
};

Binlog_transmit_observer transmit_observer = {
    sizeof(Binlog_transmit_observer), // len

    repl_semi_binlog_dump_start, // start
    repl_semi_binlog_dump_end,   // stop
};
```

```
repl_semi_reserve_header, // reserve_header
repl_semi_before_send_event, // before_send_event
repl_semi_after_send_event, // after_send_event
repl_semi_reset_master, // reset
};
```

现在看来,一个插件就全部对上了,如果哪天想自己实现一个 Observer 中的小功能插件,也知道如何做了。

插件安装

一个插件,可以通过如下语句安装。

```
 INSTALL PLUGIN repl_semi_sync_master SONAME 'semisync_master.so';
```

在安装之后,如果数据库重启了,以后就不需要再次安装,因为这个信息已经被存储到数据库表 `mysql.plugins` 中了。数据库启动时,系统会自动从这里找到所有的插件,自动安装。

半同步自动开关

半同步自动关闭,有以下 3 种情况。

- 当参数 `repl_semi_sync_master_wait_no_slave` 为 OFF,主库在做 `transmit_stop` 时,此时正常复制的从库个数小于设置的 `repl_semi_sync_master_wait_for_slave_count` 个数,此时主库会将其半同步关闭。
- 当主库等待从库反馈消息时间超过设置的超时时间后,主库将自动关闭半同步。
- 当数据库服务器正常退出时,半同步会先自动关闭,不过此时有可能正在等待从库的反馈消息,退出之后,有可能造成一部分事务的丢失。

半同步自动开启,只有下面一种情况。

当收到从库反馈消息之后,如果满足了通知主库的条件,同时当前最大的可提交 Binlog 的位置已经大于当前正在提交的事务中最大的 Binlog 位置,那就说明从库已经赶上了主库的进度,此时主库就自动恢复半同步复制。

15

MySQL 5.7 多线程复制原理

背景

MySQL 通过 Binlog 进行主从复制，一直是用户爱恨交加的一个实现方式。所谓爱，在于它维护容易、分析简单且架构设计可以变化多端，这在使用 MySQL 的过程中，可以发挥 DBA 的想象来解决各种各样的问题，所以受到了业界朋友的青睐。说到恨，有一个问题很是令 DBA 头疼，即主从复制延迟的问题。一般在问题出现时，DBA 只能看着，一脸茫然，无法下手，只能静静地等着它追上来（当然也有一些方法，可以适当地提升其速度，但一般都是补救，不能将速度一下子提升几倍之多），这时 DBA 可能就会对它“恨铁不成钢”了吧。

行之有效的延迟优化方法

那么在延迟的时候，如何适当地提升速度呢？一般有如下这些方式。

- 增大从库参数 `innodb_buffer_pool_size` 的值，可以缓存更多数据，减少由于转换导致的 IO 压力。
- 增大参数 `innodb_log_file_size`、`innodb_log_files_in_group` 的值，减少 BUFFER POOL 的刷盘 IO，提升写入性能。
- 修改参数 `innodb_flush_method` 为 `O_DIRECT`，提升写入性能（在 `ssd` 下，或者磁盘 IO 能力强的时候推荐使用）。
- 如果可以的话，把从库 Binlog 关掉，或者关掉参数 `log_slave_updates`。

- 修改参数 `innodb_flush_log_at_trx_commit` 为 0 或 2。
- 如果 Binlog 没有关掉，修改 `sync_binlog` 参数为 0 或一个很大的数，减少磁盘 IO 压力。
- 如果 `binlog_format` 为 ROW 模式，并且被修改表没有主键，则需要加上主键。
- 如果 `binlog_format` 为 ROW 模式，则可以在从库中删掉一些不必要的索引（同步完毕之后再加上）。
- 了解清楚写库上的操作内容，适当地在从库中预热一下数据，可以减少在复制时等待的时间。
- 如果 `binlog_format` 为 STATEMENT 模式，或者存在 DDL 复制，则可以将 `tmpdir` 参数改到内存中，比如 `/dev/shm`。
- 修改参数 `master_info_repository`、`relay_log_info_repository` 为 TABLE，减少直接 IO 导致的磁盘压力。
- 将从库的服务迁走，当然这是指简单的处理，比如使 VIP 飘到其他节点上等。
- 升级硬件，这种方法比较暴力，但一般情况下不太实际。
- 如果当前版本是 MySQL 5.6 版，并且实例中数据库比较多，写入比较均匀，则可以打开多线程复制。
- 升级成 MySQL 5.7 版吧。

MySQL 5.6 的多线程复制

在上面提到的优化方法中，有一种方法是如果当前版本是 MySQL 5.6 版，并且实例中数据库比较多，写入比较均匀，则可以打开多线程复制。很明显可以看出，使用 MySQL 5.6 版是有条件的，如下两点。

- 当前实例中涉及的数据库比较多。
- 每个数据库的数据写入比较均匀。

但在一般的 MySQL 使用中，一库多表比较常见，而多库少表比较少见。单单是这一点，就把很多人希望通过升级 5.6 的多线程复制来解决主从延迟的想法给抹杀了。

另外，要求写入比较均匀这点也比较少见。多个库同时都在写入的情况下，可能在设计数据库之初，需要为了迎合 MySQL 5.6 按库来分的特点来设计库表结构。当然，这里的意思还是说，真正能用 MySQL 5.6 的多线程复制来减少延迟的情况还是少之又少。

这里可能会有人问，如果出现了跨库的情况，怎么办？关于这个问题，MySQL 能给出的答案就一个字：等。

又有人问, MySQL 5.6 为什么要把复制分发级别设置为库, 而不是表。这个, 本人考虑可能是官方担心有跨表修改等操作会导致冲突太多, 可能实际效果不是太好。或者在 MySQL 看来, 不管是按库分, 还是按表分, 都是一样的。多个表放在一个库中, 或者一个表对应一个库, 都没什么区别。而 MySQL 只是实现了按库来分, 或许是对用户使用方式的一种引导吧。总之, 现在已经有了 5.7 版, 其他话就不多说了。

MySQL 5.7 的多线程复制

上面已经讲到, MySQL 5.6 表面上已经支持并行复制了, 但实际上属于“雷声大雨点小”。这种支持并行复制的功能出来之后, 很多人都觉得不太适合, 所以使用的人比较少。在沉寂了一段时间之后, MySQL 5.7 的 beta 版本出来了, 它的并行复制以一种全新的姿态出现在了 DBA 面前, 每个人都叫好。从此之后, 所有人都把解决复制延迟的希望, 放在了 MySQL 5.7 版本上。这也所言非虚, MySQL 这次是来真的了。很多人也说, 这才是真正原汁原味的并行复制, 并行复制本来就应该做成这个样子。

那就来看看 MySQL 5.7 中的并行复制究竟是如何实现的。

首先, MySQL 5.7 的并行复制基于一个前提, 即所有已经处于 prepare 阶段的事务, 都是可以并行提交的。这些当然也可以在从库中并行提交, 因为处理这个阶段的事务, 都是没有冲突的, 该获取的资源都已经获取了。反过来说, 如果有冲突, 则后来的会等已经获取资源的事务完成之后才能继续, 故而不会进入 prepare 阶段。这是一种新的并行复制思路, 完全摆脱了原来一直致力于为了防止冲突而做的分发算法、等待策略等复杂而又效率低下的工作, 有一种“山重水复疑无路, 柳暗花明又一村”的感觉。

根据以上描述, 这里的重点是如何来定义哪些事务是处于 prepare 阶段的, 以及在生成的 Binlog 内容中该如何告诉 SLAVE 哪些事务是可以并行复制的。MySQL 5.7 为了兼容 5.6 版的库级复制, 增加了一个参数 `slave_parallel_type`, 用来与之前 5.6 版的库级复制区分。5.6 版的库级复制参数值为 `DATABASE`, 而 5.7 版的并行复制参数值为 `LOGICAL_CLOCK`。

首先来看一下 5.7 版中生成的 Binlog 内容, 如图 15.1 所示。

图 15.1 中所示的只是将 GTID 事件过滤了出来, 其他的和以前版本是一样的。可以看出, GTID 这个事件相比 5.6 版本, 多了如下两个内容。

- `last_committed`。
- `sequence_number`。

如图 15.1 所示, `last_committed` 有三个值, 分别是 0、1、4, 这就表示当前 Binlog 包括三个组。也就是说, `last_committed` 中的每个值对应于一个组的编号。`last_committed` 为 4 (1) 的有三

个事务,这三个事务在 5.7 版本中,就被定义为可以并行复制(提交)的,而 `sequence_number` 是顺序增长的,每一个事务对应一个序列号(`sequence_number`)。

```
#160325 11:19:12 server id 56711 end_log_pos 339 CRC32 0x2280d22e GTID last_committed=0 se
quence_number=1
#160325 11:19:41 server id 56711 end_log_pos 494 CRC32 0x0d92eae0 GTID last_committed=1 se
quence_number=2
#160325 11:19:44 server id 56711 end_log_pos 762 CRC32 0xfc12c5e1 GTID last_committed=1 se
quence_number=3
#160325 11:19:45 server id 56711 end_log_pos 1031 CRC32 0xd314c365 GTID last_committed=1 se
quence_number=4
#160325 11:22:23 server id 56711 end_log_pos 1300 CRC32 0x8dfca123 GTID last_committed=4 se
quence_number=5
#160325 11:22:24 server id 56711 end_log_pos 1568 CRC32 0x8e5cb7f8 GTID last_committed=4 se
quence_number=6
#160325 11:22:25 server id 56711 end_log_pos 1837 CRC32 0x0e415e3f GTID last_committed=4 se
quence_number=7
```


图 15.1

另外,还有一个细节可能不太容易被发现,其实每一个组的 `last_committed` 值,都是上一个组中事务的 `sequence_number` 最大值,也是本组中事务 `sequence_number` 最小值减 1。同时这两个值的有效作用域都在文件内,只要换一个文件(flush binary logs),这两个值就都会从 0 开始计数。

那么此时,还有一个很重要的技术问题——MySQL 是如何做到将这些事务分组的呢?要搞清楚这个问题,首先需要了解一下 MySQL 的提交方式——`ordered_commit`。

ordered commit

关于这部分的处理,MySQL 使用了一个函数来处理,如下。

```
 int MYSQL_BIN_LOG::ordered_commit(THD *thd, bool all, bool skip_commit)
```

先看它的逻辑图,如图 15.2 所示。

从图 15.2 中可以看到,只要事务提交(调用 `ordered_commit`),就都会先加入队列中。而提交有三个步骤,包括 FLUSH、SYNC 及 COMMIT,相应地也有三个队列。首先要加入的是 FLUSH 队列,如果某个事务加入时,队列还是空的,则这个事务就担任队长,来代表其他事务执行提交操作。而在其他事务继续加入时,就会发现此时队列已经不为空了,那么这些事务就会等待队长帮它们完成提交操作。在图 15.2 中,事务 2~6 都是这种坐享其成之辈,事务 1 就是队长了。不过这里需要注意一点,不是说队长会一直等待要提交的事务不停地加入,而是有一个时限,只有在这个时限之内成功加入到队列的,才能帮它提交。这个时限就是从队长加入开始,到它去处理队列的时间,这个时间实际上非常小,基本上就是程序从这行到那行的一个过程,也没有刻意去等待。

只要队长将这个队列中的事务取出,其他事务就可以加入这个队列了。第一个加入的还是队长,但此时必须要等待。因为此时有事务正在做 FLUSH,做完 FLUSH 之后,其他的队长

才能带着队员做 FLUSH。而在同一个时刻，只能有一个组在做 FLUSH。这就是图 15.2 中所示的等待事务组 2 和等待事务组 3，此时队长会按照顺序依次做 FLUSH，做 FLUSH 的过程中，有一些很重要的事务需要去做，如下。

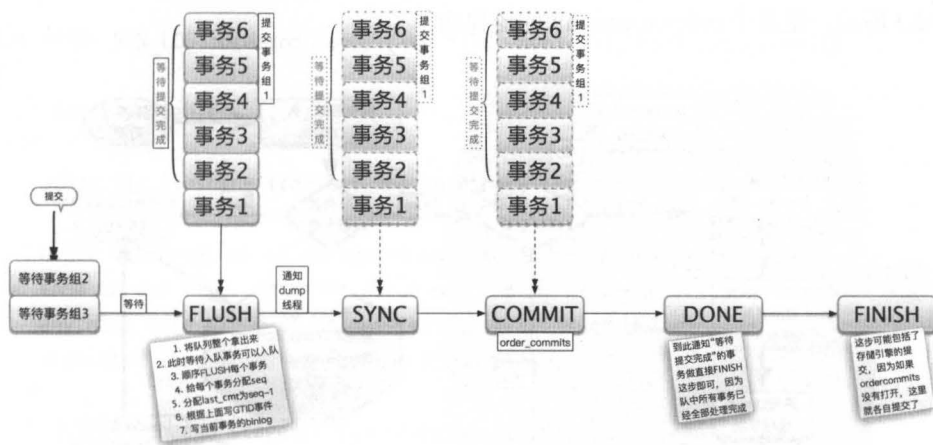


图 15.2

1. 要保证顺序必须是提交加入到队列的顺序。
2. 如果有新的事务提交，此时队列为空，则可以加入到 FLUSH 队列中。不过，因为此时 FLUSH 临界区正在被占用，所以新事务组必须要等待。
3. 给每一个事务分配 `sequence_number`，如果是第一个事务，则将这个组的 `last_committed` 设置为 `sequence_number-1`。
4. 将带着 `last_committed` 与 `sequence_number` 的 GTID 事件 FLUSH 到 Binlog 文件中。
5. 将当前事务所产生的 Binlog 内容 FLUSH 到 Binlog 文件中。

这样，一个事务的 FLUSH 就完成了。接下来，依次做完组内所有事务的 FLUSH，然后做 SYNC。如果 SYNC 的临界区是空的，则直接做 SYNC 操作，而如果已经有事务组在做，则必须要等待。同样地，做完 FLUSH 之后，FLUSH 临界区会空闲出来，那么此时在等待这个临界区的组就可以做 FLUSH 操作了。总而言之，每一个步骤都会有事务组在做，就像一个流水线一样。完成一件产品需要三个工序，每个工序都可以批量来做，那么每个工序车间都不会闲着，都一直重复着相同的事情，最终每个产品都是以完全相同的顺序完成。

到 COMMIT 这道工序时，实际做的是存储引擎提交，参数 `binlog_order_commits` 会影响提交行为。如果设置为 ON，那么此时提交就变为串行操作了，就以队列的顺序为提交顺序。而如果设置为 OFF，提交就不会在这里进行，而会在每个事务（包括队长及队员）做 `finish_commit` (FINISH) 时各自做存储引擎的提交操作。组内每个事务做 `finish_commit` 是在队长完成 COMMIT 工序之后进行，到步骤 DONE 时，便会唤醒每一个等待提交完成的事务，

告诉它可以继续了,那么每个事务就会去做 finish_commit。而后,自己再去做 finish_commit。这样,一个组的事务就都按部就班地提交完成了。现在也可以知道,与这个组中同时在做提交的,最多还有另外两个事务,一个是在做 FLUSH,一个是在做 SYNC。

如图 15.3 所示,是关于 ordered commit 的流程图。

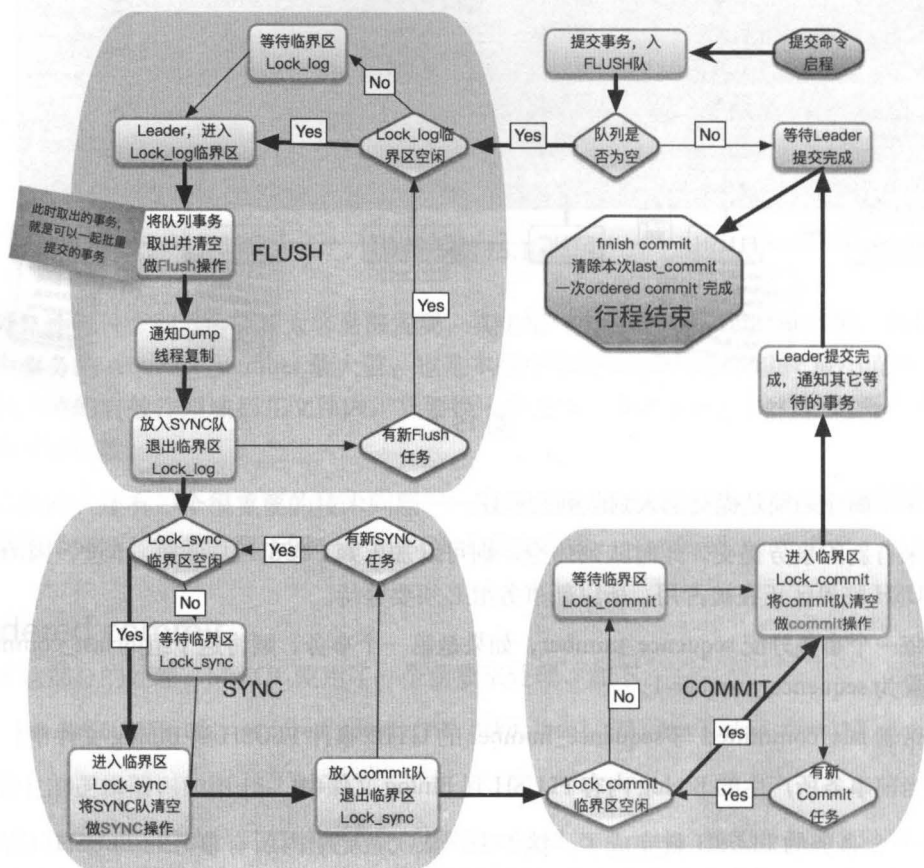


图 15.3

现在应该搞明白关于 order commit 的原理了,而这也是 LOGICAL_CLOCK 并行复制的基础。因为 order commit 使得所有的事务分了组,并且有了序列号,从库拿到这些信息之后,就可以根据序列号放心大胆地做分发了。

但是有没有发现一个问题,每一个组的事务数都没有做过特殊处理。因为从时间上说,从队长开始入队,到取队列中所有事务出来,这之间的时间是非常非常小的,其实就是几行代码的事,也不会有任何费时间的操作,所以在这段时间内其实不会有多少个事务。只有在压力很大,提交的事务非常多的时候,才会提高并发度(组内事务数变大)。不过这个问题也可

以解释得通，主库压力小的时候，从库何必要那么大的并发度呢？只有压力大的时候，从库才会出现延迟。姑且这样解释了。

再反回来看看精简之后的代码实现，这样通过对照，可以有更深的印象，如下。

```
int MYSQL_BIN_LOG::ordered_commit(THD *thd, bool all, bool skip_commit)
{
    /* local variables ... */
    /*
        Stage #1: flushing transactions to binary log

        While flushing, we allow new threads to enter and will process
        them in due time. Once the queue was empty, we cannot reap
        anything more since it is possible that a thread entered and
        appointed itself leader for the flush phase.
    */
    /* 这里调用change_stage时，返回值为true，说明当前
       事务没有做成leader，而如果为false，则说明当前事务为
       leader。如果没有做成leader，则将事务加入队列之后，
       它会一直等待，直到leader带领队列中的所有事务完成提交
       之后（即leader作为代表替非leader做提交操作），非
       leader的线程才会从这个函数返回来，然后做自己的提交
       操作，执行自己的finish_commit。不过，如果参数
       binlog_order_commits为on的话，finish_commit
       就不会做任何重要操作了，因为提交要保证顺序，要在本
       函数（ordered_commit）的最后处理当前组中的所有事务 */
    if (change_stage(thd, Stage_manager::FLUSH_STAGE, thd, NULL, &LOCK_log))
    {
        DBUG_RETURN(finish_commit(thd));
    }

    /* 当一个事务在调用以上函数并确定是leader之后，继续
       执行到这里时，这个leader就是要将这个组内的所有事务取
       出来，带领这些事务做提交操作，而在前面提到的一个时间
       窗口，就是从上面函数调用确定leader开始到这里结束。
       在这段时间内，又加入的事务，都会作为非leader的角色，
       有多少算多少，压力大的时候组内事务就会比较多，压力小
       的时候，也许每一个事务都是leader（光杆司令而已）。*/
    THD *wait_queue= NULL;
    flush_error= process_flush_stage_queue(&total_bytes, &do_rotate, &wait_queue);
```

```

/* process_flush_stage_queue执行完之后, 就可以
   知道一个组内所有的binlog的大小, 通过total_bytes
   来表示。另一个参数wait_queue, 就是表示当前组内的所有
   事务, 这是一个队列, 后面两个阶段的处理对象都是这个队列 */
my_off_t flush_end_pos= 0;
/* 如果产生了binlog内容, 则这里将它们刷入文件。注意这
   里只是将binlog内容从内存刷入文件中, 即操作系统缓存, 还
   没有做sync操作 */
if (flush_error == 0 && total_bytes > 0)
    flush_error= flush_cache_to_file(&flush_end_pos);

/*
   If the flush finished successfully, we can call the after_flush
   hook. Being invoked here, we have the guarantee that the hook is
   executed before the before/after_send_hooks on the dump thread
   preventing race conditions among these plug-ins.
*/
/* 正如上面注释所述, 如果上面写文件成功了, 则这里会执行
   signal_update函数, 这是用来通知dump线程的, 让它知道
   已经产生了新的binlog, 它可以继续向从库发送binlog了 */
if (flush_error == 0)
{
    signal_update();
}

/*
   Stage #2: Syncing binary log file to disk
*/
/* 上面所执行的, 就是第一阶段, 现在开始的是第二
   阶段, 即Sync。这里所做的操作是, 将队列wait_queue从FLUSH
   阶段转到SYNC阶段, 如果SYNC队列已经有事务在用了, 就需要
   等待, 而如果是空的, 则需要对其加锁, 并且将FLUSH队列清空, 这样就
   又可以有新的事务加入其中了 */
if (change_stage(thd, Stage_manager::SYNC_STAGE,
                 wait_queue, need_LOCK_log ? NULL : &LOCK_log, &LOCK_sync))
{
    DBUG_RETURN(finish_commit(thd));
}
/* 从SYNC队列中, 取出新加入的组队列, 用final_queue表
   示, 其实和上面的wait_queue是一样的内容。取出来之后, 如果
   有对应的Binlog内容, 则将它们对应的Binlog文件刷盘落地,

```

```

        这样Sync阶段就做完了 */
    THD *final_queue= stage_manager.fetch_queue_for(Stage_manager::SYNC_STAGE);
    if (flush_error == 0 && total_bytes > 0)
    {
        std::pair<bool, bool> result= sync_binlog_file(false);
        flush_error= result.first;
    }

    /*
    Stage #3: Commit all transactions in order.
    This stage is skipped if we do not need to order
    the commits and each thread have to execute the
    handlerton commit instead. However, since we are
    keeping the lock from the previous stage, we
    need to unlock it if we skip the stage.
    */
    /* 如上面注释所述, 这是最后一个阶段, 即提交阶段。不过, 这与
    参数binlog_order_commits有关系, 如果其值为ON, 则说明
    要顺序提交, 处理方法与上面两个阶段一样, 即先转为Commit
    阶段, 如果Commit队列被其他组占着, 就等待, 如果为空, 则
    将队列final_queue放入Commit队列中, 然后取出并用参数
    commit_queue表示, 再逐个做顺序处理。而如果参数为OFF,
    则提交操作就不是以组为单位了, 是每个事务都各自负责自己的
    提交工作, 即调用finish_commit函数。*/
    if (opt_binlog_order_commits)
    {
        if (change_stage(thd, Stage_manager::COMMIT_STAGE, final_queue, &LOCK_sync,
                        &LOCK_commit))
        {
            DEBUG_RETURN(finish_commit(thd));
        }
        THD *commit_queue= stage_manager.fetch_queue_for(Stage_manager::COMMIT_STAGE);

        process_commit_stage_queue(thd, commit_queue);
        mysql_mutex_unlock(&LOCK_commit);

        process_after_commit_stage_queue(thd, commit_queue);
        final_queue= commit_queue;
    }
    else
        mysql_mutex_unlock(&LOCK_sync);

```

```

/* Commit done so signal all waiting threads */
stage_manager.signal_done(final_queue);

/*
   Finish the commit before executing a rotate, or run the risk of a
   deadlock. We don't need the return value here since it is in
   thd->commit_error, which is returned below.
*/
/* 各自负责自己的提交, 在函数finish_commit里面会做判断。如果上面说的参数
   binlog_order_commits为ON, 则里面不会做任何操作;
   如果为OFF, 就会做完整的事务提交工作 */
(void) finish_commit(thd);

DEBUG_RETURN(thd->commit_error);
}

```

多线程复制分发原理

知道了 order commit 原理之后, 现在应该可以很容易想到, 在从库端是如何分发的。从库是以事务为单位做 APPLY 的, 每一个事务有一个 GTID 事件, 从而都有一个 last_committed 及 sequence_number 值, 分发原理如下。

1. 从库 SQL 线程拿到一个新事务, 取出 last_committed 及 sequence_number 值。
2. 判断当前 last_committed 是不是大于当前已经执行的 sequence_number 的最小值 (low water mark, 下面称 lwm)。
3. 如果大于, 则说明上一个组的事务还没有完成。此时等待 lwm 变大, 直到 last_committed 与 lwm 相等, 才可以继续。
4. 如果小于或等于, 则说明当前事务与正在执行的组是同一个组, 不需要等待。
5. SQL 线程通过统计, 找到一个空闲的 worker 线程, 如果没有空闲的, 则 SQL 线程转入等待状态, 直到找到一个为止。
6. 将当前事务打包, 交给选定的 worker, 之后 worker 线程会去 APPLY 这个事务, 此时的 SQL 线程就会处理下一个事务。



说明: 当然, 上面的步骤是以事务为单位介绍的, 其实实际处理中还是一个事件一个事件地分发。如果一个事务已经选定了 worker, 而新的 event 还在那个事务中, 则直接交给那个 worker 处理即可。

从上面的分发原理来看，同时执行的都是具有相同 `last_committed` 值的事务，不同的只是后面的需要等前面做完了才能执行，这样的执行方式有点如图 15.4 所示。

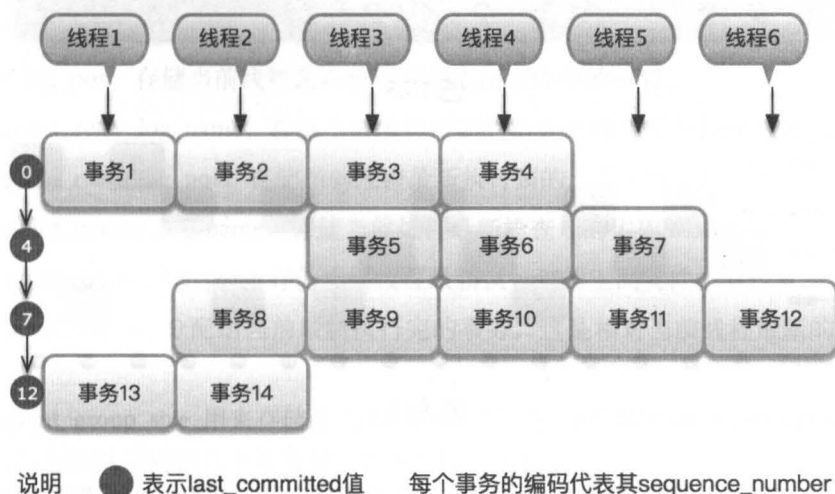


图 15.4

可以看出，事务都是随机分配到了 worker 线程中，但是执行的话，必须是一行一行地执行。一行事务个数越多，并行度越高，也说明主库瞬时压力越大。

异常故障恢复

上面已经讲清楚了 MySQL 5.7 版中的并行复制原理。但是，并行总是存在一个不可避免的问题，那就是在从库并行执行的过程中，如果数据库或操作系统挂了，那么此时每个线程执行的点就都是不确定的。也就是说，顺序的 Binlog 在被分发出去之后，从最小位置到最大位置之间这块连续的内容之间是存在断点的。如此一来，从库恢复之后，开始执行时就需要准确无误地还原哪些已经执行，哪些还没有执行，这是这节需要解决的问题。

图 15.5 所示的情况，就是一个正在执行并行执行的事务队列，也可以说这些事务的 `last_committed` 都是相同的，并且从前到后。此外，`sequence_number` 是顺序增长的，也就是说图中所示的是以 Binlog 文件内容的顺序排列的。

图 15.5 所表示的就是整个队列被分配之后，在某一时刻，队列被执行的状态，假设此时从库挂了，那么再次启动之后，如何继续执行，下面就这个问题做详细描述。

MySQL 为了实现对执行状态的记录，做了很多工作，首先就是维护一个队列，这个队列叫 GAQ (Group Assigned Queue)。SQL 线程在分配某一个事务时，首先会将这个事务加入到这个队列（队列如图 15.5 所示），之后，系统会将当前事务分发到一个线程来执行。可以想到，在某一个时刻，任务队列 GAQ 及每个线程的执行队列如图 15.6 所示。

一个Binlog事务队列，不同编号
表示由不同线程来APPLY

—已执行
—未执行



图 15.5

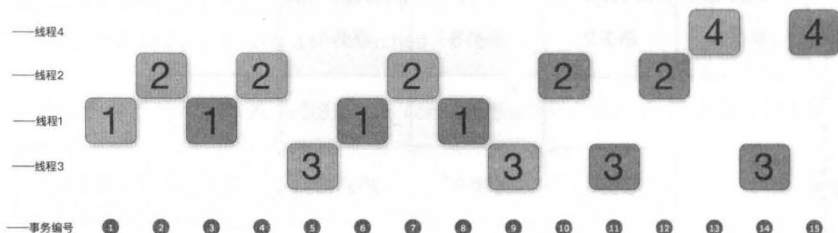


图 15.6

每一个事务，在分发之后，都会有一个编号。这个编号在某一段时间内，都是相对固定的，如图 15.6 所示，每一个事务都有一个编号，编号只要被分配了，就不会再变。在事务获取编号且被所属线程执行之后，它的位置信息会被 FLUSH 一次，这与 MySQL 5.5 版本中的 relay_info 是类似的，用来存储从库执行的位置。但现在已经变为多线程了，那么每个线程执行到什么位置，是需要记录下来的，此时就用另一个表 mysql.slave_worker_info 来存储。



```
CREATE TABLE `slave_worker_info` (
  `Id` int(10) unsigned NOT NULL,
  `Relay_log_name` text CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  `Relay_log_pos` bigint(20) unsigned NOT NULL,
  `Master_log_name` text CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  `Master_log_pos` bigint(20) unsigned NOT NULL,
  `Checkpoint_relay_log_name` text CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  `Checkpoint_relay_log_pos` bigint(20) unsigned NOT NULL,
  `Checkpoint_master_log_name` text CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  `Checkpoint_master_log_pos` bigint(20) unsigned NOT NULL,
  `Checkpoint_seqno` int(10) unsigned NOT NULL,
  `Checkpoint_group_size` int(10) unsigned NOT NULL,
  `Checkpoint_group_bitmap` blob NOT NULL,
  `Channel_name` char(64) NOT NULL COMMENT 'The channel on which the slave
    is connected to a source. Used in Multisource Replication',
  PRIMARY KEY (`Channel_name`, `Id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 STATS_PERSISTENT=0 COMMENT='Worker Information'
```

这个表，用来存储每一个线程的执行情况，有多少个线程，就有多少条记录，下面分别解释一下每个列的意义，如下。

- Relay_log_name, 存储当前线程最新执行到的从库 Relay Log 的文件名。
- Relay_log_pos, 存储当前线程最新执行到的上面文件中的位置。
- Master_log_name, 存储当前线程最新执行到的主库 Master Log 的文件名。
- Master_log_pos, 存储当前线程最新执行到的上面文件中的位置。
- Checkpoint_relay_log_name, 存储当前线程最新的检查点的从库 Relay Log 的文件名。
- Checkpoint_relay_log_pos, 存储当前线程最新执行到的上面文件中的位置。
- Checkpoint_master_log_name, 存储当前线程最新检查点的主库 Master Log 文件名。
- Checkpoint_master_log_pos, 存储当前线程最新执行到的上面文件中的位置。
- Checkpoint_seqno, 分配给当前最新执行完的事务, 这是相对当前线程最新的检查点而言的。
- Checkpoint_group_size, 用来存储最大的执行队列长度(对应参数 slave_checkpoint_group)。当执行线程的队列达到这个长度时, 必须要做一次检查点。
- Checkpoint_group_bitmap, 用来存储在当前线程最新检查点的上下文环境里, 这个线程已经执行的事务, 是一个位图值, 这是用来恢复数据库的一个关键值。
- Channel_name, 在多源复制时使用, 这里可以不用关注。

上面提到了检查点, 之前介绍 InnoDB 日志系统的时候也介绍过检查点, 这两个虽然都叫检查点, 但实际不是一个东西, 不过意义差不多, 下面详细讲述一下这里检查点的作用是什么。

上面说了每个线程在执行完一个事务之后, 会把执行位置信息写入 slave_worker_info 表中。但是, GAQ 队列长度是有限的, 不可能一直增长下去(否则 Checkpoint_seqno 就会一直增长下去)。所以必须要在这个队列中, 找到一个位置, 这个位置是 GAQ 的起点, 并且之前的 Binlog 都是已经执行过的, 而这个过程, 就被称为做检查点。那么, 多线程复制就可以归结为: 检查点在 GAQ 中周而复始地向前推进复制位置, 每个线程在不断地 APPLY Binlog, 并且通过 Checkpoint_group_bitmap 记录已经执行的事务与最新检查点的相对位置。

其实从这个意义上讲, 这里的检查点和日志检查点是一样的, 其共同特性如下。

- 有限的空间范围, 必须要清出一些空间来, 继续后面的工作。
- 在最新检查点之前的位置, 所有的工作都已经落地。
- 在最新检查点之后的位置, 都具有不确定性。
- 需要不断地做, 来保证操作不断地推进。

每个线程在接收到新的事务时, 可以不断地执行下去, 但总有一个位置, 它之前所有的 Binlog 都已经执行过了。从库需要记录下这个点, 以便在复制中断后, 再次开始复制时, 可以有一个合适的起点位置, 这个就需要检查点来保证。

检查点执行过程的大概步骤如下。

1. 在 GAQ 队列中,从队尾开始扫描,如果是已经执行过的事务,则直接将其从队列中删掉。
2. 一直扫描 GAQ,直到找到一个事务的状态为没有处理过,则停止扫描。
3. 上面扫描到的最后一个事务被确定为检查点的最新位置,并且被标记为 lwm (low water mark),前面在讲事务分发的时候,已经讲过这个概念了。
4. 将当前 lwm 这个事务对应的位置 (master_log_pos 及 relay_log_pos) 设置为此次检查点对应的位置。
5. 通知每个线程执行自己的检查点,也就是更新每个线程的 Checkpoint_seqno 值。关于这个步骤有这样一些说明:如果不更新,则这个值的参照对象为上一个检查点,而如果更新了,则其参照对象为当前最新检查点。因为这个原因,所以在 slave_worker_info 表中,每一个 Checkpoint_seqno 的值其实都是相对每个线程自己的最新检查点而言的,而不一定是全局的最新检查点。更新与不更新的决定权,在于这个线程有没有再去做事务的提交操作,如果提交就会更新这个值。

然后将此次检查点的结果写入复制位置表 (mysql.slave_relay_log_info) 中。这个表中的信息比较简单,和以前的版本差不多,主要就是一个总的复制位置信息。

上面已经提到,在事务分配之后,每一个事务都有一个编号。在一段时间内,这个编号是相对固定的,因为这个编号是从相对最新的一个检查点的位置开始的。它用来清楚地记录某一个事务在相对当前最新检查点位置的执行情况,比如从库挂了之后再开始复制的时候,根据这个编号决定要不要再执行这个事务,所以这个编号是至关重要的。

在这里,还存在一个问题需要注意,即分配时根据最新的检查点生成的编号,还没有执行完,而又做了一次检查点操作,没有执行完的事务的编号相对最新的检查点肯定是不对的。那么,此时该如何处理?

很简单,因为做一次新的检查点操作时,系统是知道这次操作向前推进了多少个事务的。编号都是顺序连续产生的,那么线程执行事务提交时,如果发现做了一次新的检查点操作,就用之前产生的所有编号简单地减去这次检查点推进的事务数即可,这样计算得到的结果就是这个事务相对新的检查点的位置。

对每一个线程而言,执行事务时,每一次事务的提交都会将当前线程的状态做一次 flush,存储到 slave_worker_info 表中,包括重要的 Checkpoint_master_log_name、Checkpoint_master_log_pos、Checkpoint_seqno 及 Checkpoint_group_bitmap 信息,这样就记录下了当前线程相对最新检查点的执行状态。

上面所述的过程,是一个不断重复、不断替代更新的过程。目的就是为了保证从库并行执行的完整性,有朝一日从库真的挂了,这些信息就可以派上用场了。讲到这里,应该都可以想

到故障恢复的过程是怎么做的了，下面就简单介绍一下这个过程。

前面所有的介绍，都是为从库 Crash 之后如何恢复复制而准备的。而在恢复时，系统首先要做的就是将 `slave_master_info`、`slave_relay_log_info` 及 `slave_worker_info` 表中的内容读出来，找到连接主库的信息 (`slave_master_info`)，然后再找到复制位置及线程个数 (`slave_relay_log_info`)，然后再找到每一个线程的复制信息 (`slave_worker_info`) 等，读取完毕后就可以开始恢复了。

从 `slave_relay_log_info` 读出来的位置，就是上面说过的系统整体最新检查点的位置。这个位置以前的每个事务肯定都已经复制完成了，那么恢复也从这个位置开始。因为从这个位置开始，后面的事务有些已经执行，有些还没有执行，下面的过程就是处理这个问题了。

恢复的过程，是按不同线程逐个分析的过程。首先，如果线程最新的执行位置落后于最新检查点的位置，那么这个线程就不需要恢复了，因为这个线程执行的所有事务都是完整的。做完这层过滤之后，再分别对剩下的线程做恢复检查。

针对每一个线程，恢复过程是一样的，概括如下。

打开最新的复制点（最新检查点的位置），统计事务的个数，同时检查当前事务的位置是不是当前线程最新检查点的位置。如果相同，则说明已经扫描到这个线程执行的最后一个事务。此时，根据当前线程的 `Checkpoint_seqno` 来统计从系统检查点位置到最后更新的位置之间，有哪些事务被当前线程执行过。这里有一点需要注意，当前线程的 `Checkpoint_seqno` 位置，是相对其最新执行的检查点而言的，这个最新的检查点位置不一定是系统最新检查点的位置。所以，这里在统计时，需要转换为相对系统最新检查点的位置，来判断当前线程已经执行过哪些事务。已经执行过哪些事务，是通过当前线程的 `Checkpoint_group_bitmap` 位图列来计算的。

每个线程都做过上面的操作之后，得到一个共同的已经执行的位图集，这个位图集是相对系统最新检查点的位置计算的。这个位图表示的是，从这个最新检查点开始，哪些事务已经执行了。此时，恢复的结果已经与线程脱离了关系，恢复过程也已经结束，其他的工作，就是在复制过程中判断这个位置集是否已经执行，如果执行了就不再执行，没有执行的再去执行，仅此而已。

现在已经明确，恢复的过程其实是复制环境恢复的过程，是一个复制上下文恢复的过程，是一个通过统一每个线程不同参照对象来计算总的执行位图的过程，所有的执行还是在正常的复制过程中来处理。

在恢复环境之后，从系统最新检查点位置开始，到线程中最大位置之间的复制执行是串行的，每执行一个事务，都会更新一次 `slave_relay_log_info` 表，这样保证了在这个过程中再次挂掉时，还可以正确地恢复。只有当这些事务都执行完了，整个恢复的过程才算完成。而此时系统会将 `slave_worker_info` 表中的位置信息都清空，表示恢复已经完成，之后整个复制就可以按部就班地进行，涛声依旧了。

16

大量 MySQL 表导致服务变慢的问题

背景

有一个业务需要分 1000 个库，每一个库中都有 80 个表，总共就是 $80000 * 2$ 个文件。文件使用率还挺高，大概是 $60000 * 2$ 。

这个业务采用的高可用架构是 MMM，由于集群机器在硬件检查时发现问题，必须要换掉。于是想了一个比较简单、影响面较小的方法去解决，就是找了另外两台机器迁移过去。同时，要求这四台机器属于同一个网段，VIP（虚拟 IP 地址）在机器之间可以漂移，这样业务就不需要修改 IP 地址即可迁移，相当于两次主从切换过程。

切换方案如图 16.1 所示。

从图 16.1 中可以看到，切换过程很简单，如下三步。

1. 先将原来的写节点（db1）与一个新的节点（db3）切换成一套集群，也就是把在 db2 上面的 VIP（读流量）切换到 db3 上面，此时 db1 与 db3 组成一套新集群。
2. 接着将 db1 和 db3 的角色互换，让 db3 成为写节点，db1 成为读节点。
3. 最后，再将 db1 读节点上面的 VIP（读流量）切换到 db4 上面，此时新的集群就是 db3 和 db4，db3 为写节点，已经切换完成。

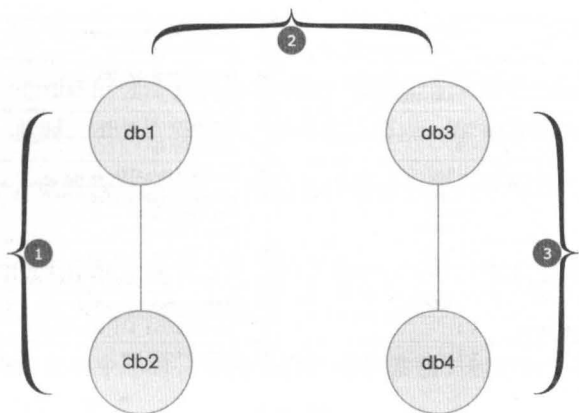


图 16.1

这样的变更是晚上做的。做好之后，观察了一段时间，发现没有什么问题（因为压力小），所以觉得事情完成了，睡吧。第二天上班之后，业务反映说迁移之后，数据库比原来慢了 10 倍（10 倍啊！感觉不可思议）。询问了一番，说没有任何变更，只是做了迁移之后就成这样了。同时经过观察，只有写库的读操作变慢了，而读库的读是不慢的。最后，业务已经受不了了，要求切换回去。还好，db1 还正在从 db3 复制，做了一个回退操作，把写挂在 db1 上面，把读挂在 db3 上面。神奇的是，问题解决了！好吧，那就先这样，走出去的路不能回头，总是要迁出去的，所以先在新旧两台机器上面挂着，查明原因后再切换回去（这样少做一步）。以上是背景。

问题分析

环境对比

- db1 写入时，db1 写不慢，读不慢，db3 也不慢。
- db3 是新的硬件，db1 是老的、有问题的硬件。
- db3 切换成写之后，在慢查询文件中明显看到很多慢查询（使用相同的语句查询，原来是 50ms，现在是 500ms），和监控是一致的。
- db1 和 db3 配置文件有差别，如图 16.2 所示（左边是 db3 的，右边是 db1 的）。

其他方面，环境完全相同，业务方面没有任何更改，重现慢的现象，只是需要切换而已。

图 16.3 是切换过程中的监控图，高起来的就是把流量切换到 db3 的情况，处于低谷的就是切换到 db1 的情况，效果非常明显，慢得立竿见影，好神奇！

character_set_server	4 FILTERED LINES	= utf8mb4
collation_server	13 FILTERED LINES	= utf8mb4_general_ci
max_allowed_packet	6 FILTERED LINES	= 32M
sql_mode	10 FILTERED LINES	= TRADITIONAL
binlog_cache_size	0 FILTERED LINES	= 1M
sync_binlog	12 FILTERED LINES	= 0
join_buffer_size	1 FILTERED LINES	= 1M
max_heap_table_size	3 FILTERED LINES	= 128M
tmp_table_size		= 128M
open_files_limit		= 150000
sort_buffer_size	16 FILTERED LINES	= 1M
innodb_flush_log_at_trx_commit	7 FILTERED LINES	= 2
innodb_open_files	6 FILTERED LINES	= 10000
innodb_print_all_deadlocks	0 FILTERED LINES	= 1
long_query_time	3 FILTERED LINES	= 1

character_set_server	4 FILTERED LINES	= utf8
collation_server	13 FILTERED LINES	= utf8_general_ci
max_allowed_packet	6 FILTERED LINES	= 128M
sql_mode	10 FILTERED LINES	= NO_AUTO_CREATE_USER
binlog_cache_size	0 FILTERED LINES	= 2M
sync_binlog	12 FILTERED LINES	= 1
join_buffer_size	1 FILTERED LINES	= 2M
max_heap_table_size	3 FILTERED LINES	= 256M
tmp_table_size		= 256M
open_files_limit		= 130000
sort_buffer_size	16 FILTERED LINES	= 2M
innodb_flush_log_at_trx_commit	7 FILTERED LINES	= 1
innodb_open_files	6 FILTERED LINES	= 0
innodb_print_all_deadlocks	0 FILTERED LINES	= 0
long_query_time	3 FILTERED LINES	= 0.1

图 16.2

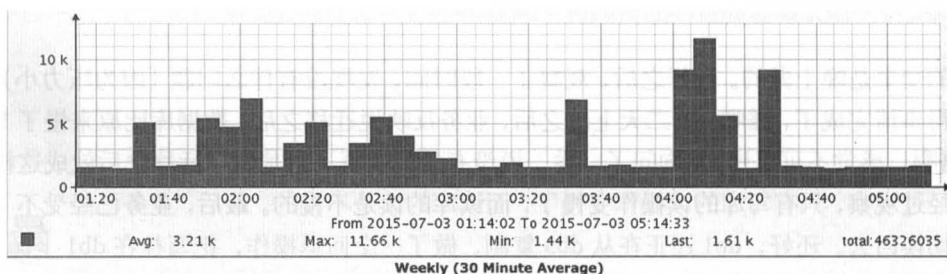


图 16.3

原因分析

- 从对比中可以得知, db1 是正常的 (以前长时间在这个机器上跑, 没有问题), 而 db3 是不正常的。这个业务目前是读多写少, 现在的现象是读慢。因为写少, 没有发现慢, 就不考虑了。
- 接着就是硬件的区别, 二者都是 PCI-e 卡, 老的、坏的概率比较大, 从经验上来看新的会比较好, 这是一个值得怀疑的点。但实际上, 针对这个问题, 找到了新卡的技术人员进行分析, 将写切换到 db3 上之后观察, 发现 IO 非常小, 能看到的监控指数都非常正常。(他们也很纳闷。)
- 除此之外, 唯一的区别就是二者的配置了, 但从图 16.3 中可以看到, 没有一个参数可以影响到让数据库的响应时间是原来的 10 倍。

但上面这些都只是分析, 硬件测试之后, 没有发现问题 (也不能说就不是硬件的问题, 一直吊在那里)。那只剩下配置了, 所以接下来从这里入手吧, 希望能成功!

那么, 再找一个夜深人静的夜晚.....

案例解决

首先要做的事情是，把 db3 的读流量切换到 db1，然后把配置完全换成 db1 的配置，将数据库重启，然后上线。此时，db1 是写节点，db3 是读节点，最神奇的时刻即将到来。

切换之后，经过观察，竟然没有问题了。问题已经解决，那么说明还是上面列出来的配置差别引起的问题。

那么解决之后，下面的工作就是重复一开始的工作，把 db1 下线，让 db4 上线。此刻，之前的迁移工作已经完成，线上服务没有问题。

但……开发同学，能给我半个小时，让我看看是哪个参数引起的么？得到的回答是：“迅速点，就这一次，给你 20 分钟。”

把最有可能的参数找出来，比如字符集（实际上，上面列出的每一个，我们认为都不会有多大影响），考虑到字符集是不可动态修改的参数，所以先把这个改了。重启，然后一个一个地动态修改、业务重启重连等，都没有发现。修改的这些参数包括：sql_mode、join_buffer_size、max_heap_size、sort_buffer_size，这些都没有影响。

结果已经说好的只有这一次，那就这样吧，任务成功完成，问题解决失败。

然而，这个问题“才下手头，却上心头”，总有一件事放心不下，约吧。和开发商量了一下，我们想解决这个问题，知道其所以然，防止在其他业务上出现同样的问题。好吧，再给你们一次机会（来之不易啊）。

那么，再找一个夜深人静的夜晚……这次的月亮好像比上次更圆一些，是好日子的征兆么？

操作之前，还简单规划了一下，下面是当时的一个计划步骤。

参数名	新配置	老配置
character_set_server	utf8mb4	utf8
collation_server	utf8mb4_general_ci	utf8_general_ci
sql_mode	TRADITIONAL	NO_AUTO_CREATE_USER
sync_binlog	0	1
join_buffer_size	1M	2M
tmp_table_size	128M	256M
sort_buffer_size	1M	2M
innodb_open_files	10000	0
innodb_flush_log_at_trx_commit	2	1

* 黑体的表示已经专门测试过，没有影响

步骤如下。

1. 考虑到先重现问题，首先应该全部使用新配置测试一次，确定问题是否还存在。
2. 因为重点考虑问题是因为 `sql_mode` 引起的，所以第二次只将这个参数改为老配置，这样就可以测试出其他配置组合时有问题，或者是没有问题，从而得出结论是 `sqlmode` 的问题。
3. 如果上面还没有找到问题原因，那么就是除了 `sql_mode` 之外的其他参数组合出了问题（如果没有，则见鬼了）。此时，通过二分法测试，先测试 `innodb_flush_log_at_trx_commit`、`innodb_open_files`、`sort_buffer_size` 三个参数。
4. 如果发现上一步有问题，则再进行二分；如果没有发现问题，则对 `sync_binlog`、`join_buffer_size`、`tmp_table_size` 进行二分。
5. 如果能走到这里，那也是醉了。
6. 再说吧。

按照步骤，一步步地开始做。

1. 首先使用有问题的配置，测试一遍，发现是老样子，还是有问题的（真是幸运，问题还存在）。
2. 把除了 `sql_mode` 之外的所有参数改成新的，其他都用老配置，测试发现没有问题。
3. 做完了，也是没有问题。
4. 做完了，还是没有问题。
5. 我醉了。

此时当事人已经搞不清楚了，难道是某两个的组合会导致出现这样的问题？如果是这样的话，那情况就太多了，天已经亮了，很累，放弃吧！

就在想放弃的时候，突然有一种新的思路。在有问题的基础上，把所有经过测试没有影响的可以动态修改的参数改成与 `db1` 相同的参数，这样应该是最少量的可以影响到性能的参数组合了。此时，在 `db1` 与 `db3` 实例上分别执行 `show variables`，全量导出变量，进行对比，发现有几个参数的区别（左边是老的 `db1`，右边是新的 `db3`），如图 16.4 所示。

binlog_cache_size	2097152	binlog_cache_size	1048576
innodb_flush_log_at_trx_commit	1	innodb_flush_log_at_trx_commit	2
innodb_print_all_deadlocks	OFF	innodb_print_all_deadlocks	ON
open_files_limit	130000	open_files_limit	150000
performance_schema_max_file_instances	200000	performance_schema_max_file_instances	230770
sync_binlog	1	sync_binlog	0

图 16.4

此时，我们做了最后的挣扎，已经只剩下这 6 个了，看上去还是不会有什么影响，有些已经试过了，再随便试一次吧。二分查找，从下面开始找了三个参数，下线、重启、上线……发现问题竟然奇迹般地存在。而此时只剩下了三个参数，其中一个参数是 `sync_binlog`，有问题的是 0，肯定不会影响啊。只能定位到剩下的两个了，可以看到倒数第二个是一个 `performance_schema` 的参数，配置文件中没有设置，是默认的，可以忽略。于是，把问题定位到 `open_files_limit` 了。

此时，再做最后一次，只剩下 `open_files_limit` 的区别了。

结果还是有问题，说明就是这个参数了。

回过头来想了一下，其实当初看区别的时候，这两个参数就在配置文件中，只是看着 13 万和 15 万相差不大，就忽略了。好吧，问题已经解决，找到了原因，天亮了，回家吧。

已经查到了是 `open_files_limit` 的原因。那么，究竟为什么在一个参数相差这么小的情况下会影响 10 倍的性能呢？查查源码！

通过 `sysbench`，创建 60000 个表，每个表 10000 行，在只读模式下，发现设置为 130000 时，QPS 可以达到 20000，而设置为 150000 的时候，QPS 只有 4000 左右。问题重现了，就简单多了。

此外，还有一个额外的发现。如果设置为 150000 之后，重启数据库，非常慢，大概需要 1 分钟，而设置为 130000 之后，只需要 10 秒左右。查看了一下在很慢的过程中 `mysqld` 的线程情况。其中，在启动的过程中，有一个线程长时间都基本处于同一个堆栈，使用 `pstack mysqldpid` 查看，如图 16.5 所示。

还有一个发现就是，当设置 `open_files_limit` 为相同的时候，`performance_schema_max_file_instances` 参数也相同了，并且这个参数没有设置过。那么，通过源码发现，这个参数竟然是通过 `open_files_limit` 值来设置的。如果 `open_files_limit` 值设置得比较大（这样就可以忽略掉其他影响条件，比如 `max_connection` 等），`performance_schema_max_file_instances` 的值直接就是从 `open_files_limit/0.65` 得来的（源码对应函数 `apply_load_factor`）。这样就知道了， $130000/0.65$ 正好是 200000， $150000/0.65$ 正好是 230770，与图 16.4 所示相符合。

另外，通过测试发现，如果单独设置 `performance_schema_max_file_instances` 为不相同的值，而将 `open_files_limit` 设置为相同，性能还是不一样。从而可以确定与 `open_files_limit` 参数实际上没有什么关系，只是 `performance_schema_max_file_instances` 使用了默认值，它的值就来源于 `open_files_limit/0.65` 了，这样间接影响了 `performance_schema_max_file_instances` 值，突然有种“隔山打牛”的感觉。

还有一个发现，如果将 `performance_schema_max_file_instances` 设置为 200000、210000、220000、230000、240000、300000 等，性能都是差不多的，唯独设置为 230770 是有问题的。仔细研究之后发现，`performance_schema_max_file_instances` 最终影响的是 `performance_schema` 数据

库中的 `file_instances` 表, 这个表中的数据是通过一个 HASH 表来缓存的, 而这个参数决定的是该 HASH 表的大小。

```
Thread 1 (Thread 0x7f3562e0e7e0 (LWP 40534)):
#0 0x000000000b45046 in lfind ()
#1 0x000000000b4516e in linsert ()
#2 0x000000000b45adc in lf_hash_insert ()
#3 0x000000000901e9b in find_or_create_file(PFS_thread*, PFS_file_class*, char const*, unsigned int, bool) ()
#4 0x000000000926795 in end_file_open_wait_and_bind_to_descriptor_v1 ()
#5 0x000000000aedic5b in fil_open_single_table_tablespace(bool, bool, unsigned long, unsigned long, char const*, char const*) ()
#6 0x000000000ad4f52 in dict_check_tablespaces_and_store_max_id(dict_check_t) ()
#7 0x000000000a4a7af in innobase_start_or_create_for_mysql() ()
#8 0x000000000988104 in innobase_init(void*) ()
#9 0x00000000005af928 in ha_initialize handlerton(st_plugin_int*) ()
#10 0x00000000006f93e1 in plugin_initialize(st_plugin_int*) ()
#11 0x00000000006fd5b6 in plugin_init(int*, char**, int) ()
#12 0x00000000005a2742 in init_server_components() ()
#13 0x00000000005a76e5 in mysqld_main(int, char**) ()
#14 0x0000003ee61ecdd in __libc_start_main () from /lib64/libc.so.6
#15 0x0000000000598bf1 in _start ()
```

图 16.5

后面又做了一个非常无聊的测试, 是 `performance_schema_max_file_instances` 值与 QPS 的对比, 如图 16.6 所示。

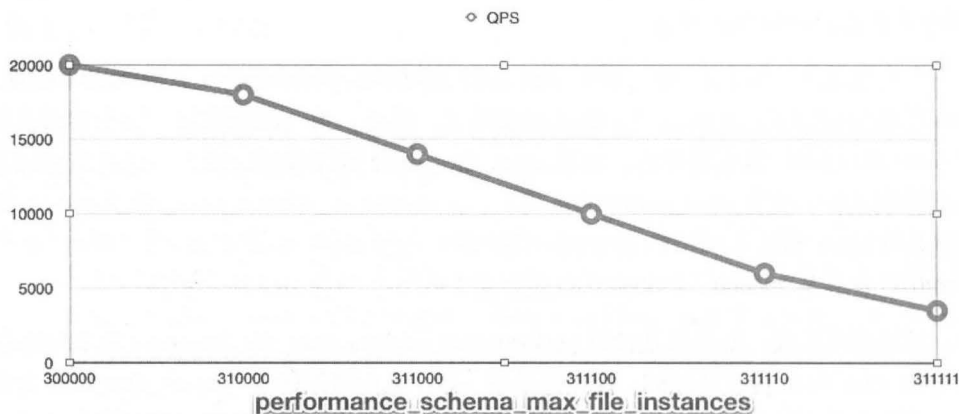


图 16.6

至此, 一切都豁然开朗了。结合上面非常慢的堆栈, 以及将 `performance_schema_max_file_instances` 设置为不同值的现象, 可以确定, 这个问题最终是 HASH 算法的问题。当 HASH 桶大小为在一个比较好看的数值时, 这个算法就非常快, 而如果是一个比较零碎的值时, 算法就非常慢了, 会导致响应时间是原来的 10 倍。

总结

- 这个问题，确实很诡异，万万没有想到是相差那么小的一个变量的问题（以至于一开始就被忽略了）。
- 这个问题，比较少见，只有表比较多时候才会比较明显（因为表比较少的时候，算法相对比较稳定）。
- 这个问题，查明了，其实就是一个算法方面的 BUG。
- 这个问题，简单的规避方法是单独设置 `performance_schema_max_file_instances` 值为 0，或者设置 `performance_schema_max_file_instances` 为一个比较好看的数值，又或者设置 `open_files_limit` 为 0.65 的整数倍，这样都不会有问题。
- 这个问题，虽然影响比较小，但我们对问题的探索精神不能没有，要了然于胸。
- 这个问题，到此为止。

17

MySQL 快速删除大表

背景

删除表，对于 MySQL 来说是一件习以为常的事情。在业务下线后，或者业务不再需要某张表时，数据库管理人员就需要清理这些废弃的数据库表等。遇到这种情况，要直接删除表吗？如果表过大，那么直接删除对 MySQL 实例是否存在影响？带着这些问题，将会从源码层面剖析删除可能带来的问题，以及如何快速删除表。

问题分析

一天早上接到一个需求，业务下线了，数据库中的某张表不需要了。为了减少磁盘大小的消耗，提出了将表删除的需求，面对这样的需求，DBA 肯定是支持的。于是，登录到 MySQL 实例上执行了 DROP TABLE 的操作，消耗了大约 35 秒。这时候，发现 MySQL 夯住了。通过监控发现，数据库的 QPS 瞬间下来很多。发现了问题，就需要来排查问题。首先，确定了一下表的大概大小，发现表的数据量达到了 200G。难道是因为表数据量太大导致的？从源码看看吧，看一下 DROP TABLE 操作、MySQL 底层干了些什么？

MySQL 在删除表的时候，主要分为以下两个过程。

1. Buffer Pool 页面清除过程。在删除表的时候，InnoDB 会将文件在 Buffer Pool 中对应的页面清除。对于删除表的页面清除，只需要将页面从 flush 队列中删除即可，而不需要去做 flush 操作，这样可以减小对系统的冲击。

2. 删除 ibd 磁盘文件的过程。

具体到对 Buffer Pool 的影响, 删除线程首先会根据要删除表的 space id, 从 Buffer Pool 中每一个 Buffer Pool 实例的 flush list 中找到属于被删除表的页面。在每个实例中搜索页面时, 会持有各自 Buffer Pool 实例的锁, 然后遍历搜索这个 Buffer Pool 实例, 如果找到了对应的页面, 就会将这个页面从 flush list 中删除, 并且将其 oldest_modification 设置为 0, 用来表示这个页面已经失效。不过, 这个操作只是将其从 flush list 中删除了, 它还会在 Buffer Pool 的空闲池中存在, 以便重新使用。

这里的问题就是, 如果 Buffer Pool 很大, 或者是在 Buffer Pool 中有很多需要被 flush 的页面, 那么此时遍历扫描页面时就会占用比较长的时间, 导致其他事务在用到相应 Buffer Pool 实例时被阻塞, 从而影响整个数据库的性能。

下面精简后的代码讲述的是每一个 Buffer Pool 实例在删除页面时的方法。

```

buf_flush_dirty_pages(
    buf_pool_t* buf_pool,    /*!< buffer pool instance */
    uint      id,           /*!< in: space id */
    FlushObserver* observer, /*!< in: flush observer */
    bool      flush,        /*!< in: flush to disk if true otherwise
                             remove the pages without flushing */
    const trx_t* trx)       /*!< to check if the operation must
                             be interrupted */
{
    dberr_t    err;

    do {
        /* 获取Buffer Pool mutex */
        buf_pool_mutex_enter(buf_pool);

        /* 从当前指定的Buffer Pool实例buf_pool中, 删除指定表空间的所有页面 */
        err = buf_flush_or_remove_pages(
            buf_pool, id, observer, flush, trx);

        /* 释放buffer pool mutex */
        buf_pool_mutex_exit(buf_pool);
        /* other code */
    }
}

```

从代码中看出, 针对每一个在 Buffer Pool 实例中的具体操作, 调用了函数 buf_flush_or_remove_pages, 具体实现如下。

```

buf_flush_or_remove_pages(
    buf_pool_t* buf_pool,    /*!< buffer pool instance */
    uint      id,           /*!< in: target space id for which
                             to remove or flush pages */
    FlushObserver* observer, /*!< in: flush observer */
    bool      flush,        /*!< in: flush to disk if true but
                             don't remove else remove without
                             flushing to disk */
    const trx_t*  trx)      /*!< to check if the operation must
                             be interrupted, can be 0 */
{
    buf_page_t* prev;
    buf_page_t* bpage;
    uint      processed = 0;

    /* 获取 buffer flush list mutex */
    buf_flush_list_mutex_enter(buf_pool);

rescan:
    bool    all_freed = true;

    for (bpage = UT_LIST_GET_LAST(buf_pool->flush_list); //循环遍历
         /* other code */
         /* 这里通过对bpage->id.space()的判断, 来确定是不是要清除的页面。
            如果不是, 就直接跳过, 继续处理下一个; 如果是, 就通过函数
            buf_flush_or_remove_page来处理 */
         if ((observer != NULL && observer != bpage->flush_observer)
             || (observer == NULL && id != bpage->id.space())) {

        /* Skip this block, as it does not belong to
           the target space. */
    } else if (!buf_flush_or_remove_page(buf_pool, bpage, flush)) {
        all_freed = false;
    } else if (flush) {
        goto rescan;
    }

    ++processed; /* 计数 */

    /* 比较processed 与 BUF_LRU_DROP_SEARCH_SIZE 大小并作出相应处理 */
    if (buf_flush_try_yield(buf_pool, prev, processed)) {

```

```

        /* Reset the batch size counter if we had to yield. */

        processed = 0;
    }
    /* other code */
}

buf_flush_list_mutex_exit(buf_pool); /* 释放buffer flush list mutex */

return(all_freed ? DB_SUCCESS : DB_FAIL);
}

```

通过上面的源码可以看出 MySQL drop table 的过程如下。

- 通过 `buf_pool_mutex_enter(buf_pool)` 函数持有 buffer pool mutex。
- 通过 `buf_flush_list_mutex_enter(buf_pool)` 函数持有 buffer pool 中的 flush list mutex。
- 开始扫描 flush list 列表。
 - 如果脏页属于 DROP TABLE，则直接从 flush list 列表中移除。
 - 如果占用 CPU 和 mutex 时间过长，则调用 `buf_flush_try_yield` 函数释放 CPU 资源、flush list mutex 和 buffer pool mutex，并调用 `os_thread_yield()` 函数强制进行 context switch。
 - 重新持有 buffer pool mutex。
 - 重新持有 flush list mutex。
- 释放 flush list mutex。
- 释放 buffer pool mutex。

在大表删除时，遍历 Buffer Pool 中的每一个实例，从 flush list 中移除页面是一个比较耗时的操作。

除此之外，在删除 ibd 文件时，如果文件过大也会带来大量的 I/O，并且耗时。对于这样的问题，该如何解决呢？可以通过下面的方案来操作。

案例解决

我们已经知道删除表分为两个过程，第一个过程涉及源码部分，优化或者解决相对困难，而对于第二个过程（删除 ibd 文件）来说，其实可以在删除 ibd 文件之前，对 ibd 文件做一个硬链接来加速删除，减少对数据库造成的影响，具体操作如下。



```

sudo ln /data/mysql/testdb/example_table.ibd
/data/mysql/testdb/example_table.ibd.hdlk

sudo ls -lh /data/mysql/testdb

-rw-rw---- 1 mysql mysql 8.4K Oct 28 13:26 example_table.frm
-rw-rw---- 2 mysql mysql 100G Oct 28 13:26 example_table.ibd
-rw-rw---- 2 mysql mysql 100G Oct 28 13:26 example_table.ibd.hdlk

```

上面的命令执行完之后,发现多了一个 `example_table.ibd.hdlk` 文件,且 `example_table.ibd.hdlk` 和 `example_table.ibd` 的 inode (关于操作系统文件引用数目的知识请参考 Linux 相关书籍) 数均为 2。因为我们知道,一个磁盘上的文件,可以由多个文件系统的文件引用,这多个文件是完全相同的,都指向同一个磁盘上的文件,当我们删除任何一个文件的时候,都不会影响真实的文件,只是会将其被引用数目减 1,只有当被引用数目变为 1 的时候,再次删除文件,才会真正被删除。删除时,这两种情况的区别很明显,一个是在减少被引用数目,一个是真正做 IO 来删除它。我们正是利用了这个特点,将由 MySQL 来删除大文件的操作转换为一个简单的操作系统级的文件删除,从而减少了对 MySQL 的影响。当然,在成功 `drop table` 之后,剩下的这个真正的文件,可以想办法慢慢处理,此时已经与 MySQL 没有关系了。

如果还是担心删除大文件对操作系统有影响,从而进一步影响到了 MySQL,还有其它方法可以用来处理,比如可以循环分块删除,慢慢地清理文件,通过一个脚本即可搞定,方法是操作系统级的文件 `truncate`,这个可以自行了解。

发散思维

在普通的 MySQL 实例上,可以通过文件的硬链接方式来解决。但是 Galera Cluster 在删除表的时候,进行的验证、复制等操作都会导致集群节点出现问题,相当于一个大事务(关于 DDL 和大事务在 Galera Cluster 上执行的问题,请参考第 29 章),这时就不能直接删除了。

该如何解决呢?众所周知,在 Galera Cluster 中有一个参数可以控制本地操作是否要复制到其他节点,那就是 `wsrep_on` 参数。该参数是会话级别的,默认值为 ON,如果设置为 OFF,则当前会话的操作不会复制到其他节点,并会像独立的 MySQL 服务器一样运行。具体操作如下。



```

# 集群中的一个节点

mysql> set @@session.wsrep_on = OFF;

Query OK, 0 rows affected (0.00 sec)

```

这时可以通过上面描述的删除大表的方法，使用硬链接的方式来删除。建立硬链接的操作
此处不做赘述。

```
mysql> drop table t2;
```

```
Query OK, 0 rows affected (0.01 sec)
```

可以在集群中的每一个节点开启一个新会话，设置参数 `wsrep_on` 为 OFF，然后分别通过上面描述的删除大表的方法来删除集群中的大表，线上数据库几乎无影响。

总结

在删除大表的时候，一定要注意对线上数据库的影响。对于大表，可以通过建立硬链接，或者先清理数据，最后再删除的方式，达到预期效果。总之，线上数据库最重要，要让所有操作尽可能小地影响数据库。


18

两条不同的插入语句导致的死锁

背景

死锁，对于 MySQL 来说，是一件习以为常的事情。有多种原因导致 MySQL 的死锁率很高，包括隔离级别、GAP 锁、索引的顺序及业务逻辑等。所以，分析死锁可能会非常困难。因为 MySQL 关于死锁提供的日志信息有限，并且不完善，所以很难重现死锁发生时的场景。但有一些死锁比较常见，而且是有迹可寻的，所以本章就通过一个死锁的案例，根据各种资源来追本溯源，找到死锁发生的真正原因，并且通过其中的一些方法，起到举一反三的作用。

下面这个案例，已经是多次碰到了，是一个比较经典的死锁案例，先来看一下发生死锁的表结构，如下。

```
 CREATE TABLE my1 (  
  sno int(11) NOT NULL DEFAULT '0',  
  name int(11) DEFAULT NULL,  
  PRIMARY KEY (sno),  
  UNIQUE KEY idx (name)  
) ENGINE=InnoDB
```

表很简单，除了主键之外，只有一个在 name 列上的唯一索引，数据库隔离级别是 REPEATABLE-READ。先来看发生死锁之后的状态图，如图 18.1 所示。

LATEST DETECTED DEADLOCK-----
140528 18:32:51

*** (1) TRANSACTION:

TRANSACTION 508, ACTIVE 6 sec inserting

mysql tables in use 1, locked 1

LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1

MySQL thread id 2, OS thread handle 0x1204, query id 20 192.168.84.1 zhufeng update

insert into my1 values(6,6)

*** (1) HOLDS THE LOCK(S):

RECORD LOCKS sequence 7 space id 0 page no 342 n bits 72 index `idx` of table `my`.`my1`

trx id 508 lock mode S waiting

Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

0: len 4; hex 80000006; asc ;;

1: len 4; hex 80000005; asc ;;

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS sequence 7 space id 0 page no 342 n bits 72 index `idx` of table `my`.`my1`

trx id 508 lock mode S waiting

Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

0: len 4; hex 80000006; asc ;;

1: len 4; hex 80000005; asc ;;

*** (2) TRANSACTION:

TRANSACTION 506, ACTIVE 19 sec inserting

mysql tables in use 1, locked 1

3 lock struct(s), heap size 1248, 2 row lock(s), undo log entries 2

MySQL thread id 1, OS thread handle 0x674, query id 21 192.168.84.1 zhufeng update

insert into my1 values(7,5)

*** (2) HOLDS THE LOCK(S):

RECORD LOCKS sequence 6 space id 0 page no 342 n bits 72 index `idx` of table `my`.`my1`

trx id 506 lock_mode x locks rec but not gap

Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

0: len 4; hex 80000006; asc ;;

1: len 4; hex 80000005; asc ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS sequence 8 space id 0 page no 342 n bits 72 index `idx` of table `my`.`my1`

trx id 506 lock_mode x locks gap before rec insert intention waiting

Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0

0: len 4; hex 80000006; asc ;;

1: len 4; hex 80000005; asc ;;

*** WE ROLL BACK TRANSACTION (1)

图 18.1

在图 18.1 中，有一部分看上去好像与通常看到的一般内容不同，需要注意，具体如下。

对于事务 1，在原生的 MySQL 中，通常见到的只有 *** (1) WAITING FOR THIS LOCK TO BE GRANTED: 部分，而没有 *** (1) HOLDS THE LOCK(S): 部分。但是为了更好地追踪并解释死锁的具体锁情况，将事务 1 已经加的锁也明确打印了出来，这是通过在 MySQL 中修改源码来实现的。

还有另一个不同之处在于，原生的 MySQL 在死锁的说明信息中，只有关于某个锁的类型，很难从中看出它们的加锁顺序。一般采用推测的办法来判断某个锁是在什么时机加上去的，所以在源码中还加入了上锁的一个序列号。从图 18.1 可以看到，RECORD LOCKS sequence 7 space id 0 中的 sequence 7 就是专门为了获取加锁顺序而存在的。先加的锁的序列号会比后加的锁的序列号小，这样便能很明确地知道加锁顺序，也更容易理解和入手分析死锁的原因。

下面，具体说一下在构造死锁时的执行顺序（注意，这个测试是在空表的情况下测试的）。

事务 2	事务 1
begin;	
@1: insert into my1 values(8,6) 插入成功	
	begin;
	@2: insert into my1 values(6,6) 阻塞...
@3: insert into my1 values(7,5) 插入成功	
	报死锁，回滚事务 1

这里有一个问题。如果事务 2 后面插入的语句变为 insert into my1 values(9,7)，那么还会死锁么？这个问题在解释了上面的死锁问题后，自然就清楚了。

问题分析

从图 18.1 的序号入手分析。

很明显可以看出，上面其实只有三个锁，序号分别是 sequence 6、7、8，所以 6 是最先加上去的，锁的类型是 LOCK (X|NOTGAP|REC)。这个表面上很容易解释，因为它插入了，所以对它插入的这个记录上了一个写的记录锁，但其实还有更深入的一层，下面再一层层解释，现在可以先这样理解。这就是上面语句 @1 所处的状态，但是要注意，刚才这个锁加在了列 name 的索引 idx 上面，而不是聚簇索引上面。

然后分析 sequence 为 7 的锁。此时发现这个锁也是加在与上面序号为 6 的锁的相同位置，只是锁的类型不同。可以看到它的锁类型是 LOCK (S|WAITING)，这里只是一个记录的 S 锁，这是为什么呢？插入之后会上 S 锁么？

具体原因是，在事务 1 做语句 @2 的操作时，首先将 RECORD (6, 6) 这条记录插入到聚簇索引中，这个索引的键是 sno，也就是 6，因为是主键，并且不冲突，所以这个记录在这个索引中可以顺利插入。

再插入第二个索引，也就是在列 `name` 上的 `idx` 索引，是一个唯一键。在插入之前，需要判断是不是已经存在这个键值。在做判断之前，事务 2 已经插入了一条记录，在 `name` 列上的值为 6，这里再插入 6 时，便会发生冲突。这里所要做的工作是检查是否冲突，所以上的是读共享锁 `S`。但在检查的时候，发现这条记录已经存在，而且发现这条记录上面的事务还是 `ACTIVE` 状态，并且不是自己，这就说明上这个锁的事务（事务 2）对于事务 1 而言，还是不可见的。此时，事务 1 会将这条已经存在的记录上面的隐式锁转换为显式锁，不过此时这个隐式锁还不存在，是通过另一种方式来实现的。

这里所谓的隐式锁，是事务 1 在发现自己需要更新的记录被其他事务（事务 2）占有了，但还没有上锁的时候，此时事务 1 会“帮忙”给那个事务（事务 2）上一个锁，锁的类型由事务 2 的占用类型决定，这里是写锁。不过需要注意的是，这里是“帮忙”，是事务 1 上的锁，但锁的拥有者是事务 2。给事务 2 上锁之后，再给自己上一个读锁，这就是为什么 `sequence 7` 这个锁对应的状态是 `S` 了。同时，因为此时这个记录上已经有了写锁（事务 1 帮事务 2 上的锁），再加一个 `S` 锁肯定是不兼容的，所以状态变为 `LOCK (S|WAITING)`。

这里解释了上面说的更深层的两个问题，总结如下两点。

- 图中所示的 `sequence 6` 这个锁不是自己上的，而是别人（事务 1）帮它上的。
- 在 `InnoDB` 中，如果插入时不冲突的话，是不会上锁的。如果其他事务发现与新插入的记录有冲突，便会帮它上锁。

上面所述就是这次死锁时，前面两个锁的状态与加锁的实现方式。

再看 `sequence 8` 这个锁。可以看到，它的锁与上面锁的位置是完全一样的，只是锁的类型变为 `LOCK (X|GAP|INSERT INTENTION|WAITING)`。这个锁加锁的过程与之前不同，首先插入聚簇索引时，它的值是 7，不冲突，可以顺利插入完成。之后插入 `name` 上面的索引 `idx` 时，它的值是 5，而现在已经存在的值是 6，不会冲突，应该是成功插入并且正常执行的，并且也不会发生死锁。那么，此时为什么插入完成了，却导致事务 1 发生了死锁呢？这个问题需从一个更深层的实现来解释了，这也是对 `GAP` 锁如何实现的一个解释。

在 `InnoDB` 中，有一个锁叫 `GAP` 锁，它的出现是为了实现隔离级别 `REPEATABLE-READ` 的，从而可以保证数据在插入时不会导致幻读。`GAP` 锁，顾名思义，是一个间隙锁，它的实现依赖于一个叫 `heapno` 的逻辑编号，数据页面上的每一个记录都对应一个 `heapno`，它的值是由插入顺序及插入位置来决定的。

下面说一下 `heapno` 的实现方式。

简单来说，它是页面内每条记录的唯一记录编号。关于页面格式，在本书第 8 章中已经详细介绍过了。我们已经知道，没有插入任何数据的页面的最初状态是，只有一个最小记录及最大记录。最小记录 `heapno` 默认为 0，最大记录 `heapno` 默认为 1，同时在页面头上记录了当

前页面总的记录条数,此时为 2 (用 PAGE_N_HEAP 存储)。heapno 是从 0 开始的,所以最开始的 heapno 为 2,如图 18.2 所示。

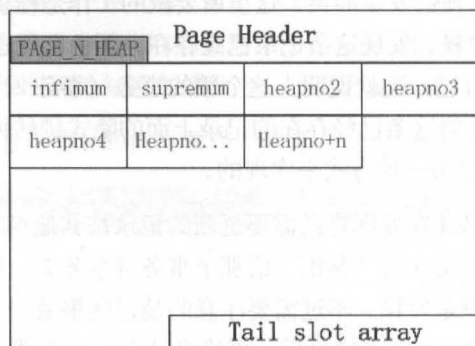


图 18.2

下面就通过两种方式来说明一下 heapno 是如何管理的。

顺序插入

如果数据是顺序插入的,也就是所有操作都是插入,并没有删除。那么,heapno 的生成就是简单按照 PAGE_N_HEAP 来计算的。因为每次插入一条记录,它也会加 1,所以 heapno 在这里一直是增加的,直到插满一个页面为止。当然,这里的插入方式不一定是追加插入,离散插入也没有问题,因为索引的顺序管理不是通过页面顺序来实现的,所以在页面内的实际物理顺序不影响索引的顺序。关于页面内记录的顺序管理方法,在第 8 章中也介绍过。

先删除后插入

一个页面在使用很久且做了很多插入、删除或更新操作之后,肯定不是像最初连续插入一样连接存储,而是有很多空洞(碎片)。而这些空间在 InnoDB 内部被称作 heap,也就是所谓的“页面堆”空间。在每 Purge 一个页面的时候,这个页面上所有的删除记录都会被清除。被清除之后腾出的空间会通过一个页面内的指针进行管理,在 InnoDB 内部被称为 PAGE_FREE。PAGE_FREE 是链表的头指针,每清除一个记录空间,它就指向这个最新的位置,并且将原来的位置放到这个空间最后面,这样最终会形成一个页面内的单链表,最新删除的都是链表前面的,而对应的记录不会被 memset,记录还是原来的记录,里面的数据不会被丢弃,最主要的是记录的 Heapno 值不会变。可以通过图 18.3 看一下上面具有空闲离散空间的页面是什么样子的。

上面已经提到,每条记录虽然被删除了,但数据还没有被清除,所以记录中的记录长度、heapno 等都不会改变,还保留了原记录中的记录信息。

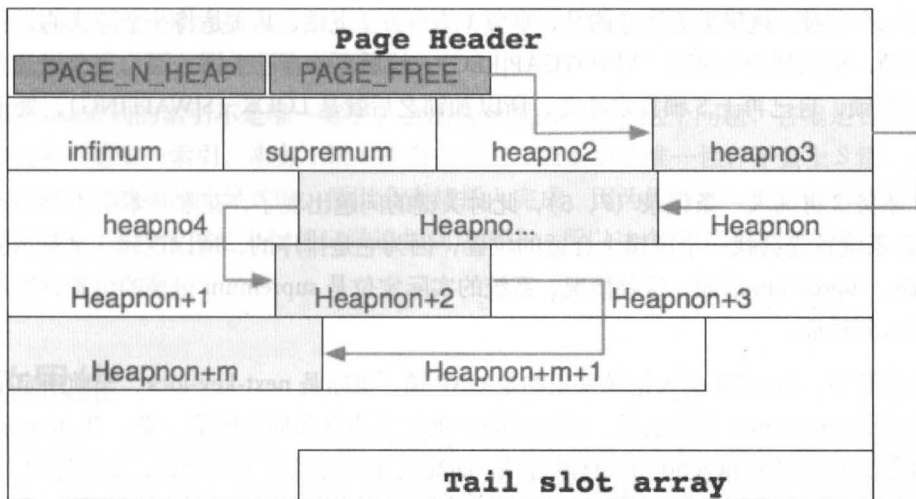


图 18.3

那么现在假设要新插入一条记录，因为所有记录在一个页面内要保证有序，所以系统首先会在逻辑上从页面槽中找到一个合适的位置。然后，在物理页上，它会试探一次当前页面的 **PAGE_FREE** 所指的空闲链表的首节点。如果发现当前的记录比空闲位置记录长度大，则直接将这个记录追加存储到页面后的堆空闲空间中去，此时的 **heapno** 是在 **PAGE_N_HEAP** 中存储的最大值，是一个新的值；如果在 **PAGE_FREE** 所指的空闲链表的首节点对应记录的空间足够存储新记录，则直接使用这个 **heapno**，然后将这个记录存储在这个位置。

那么，上面讲的就是 **heapno** 这个逻辑定义的产生及重用机制的实现。至于当 **PAGE_FREE** 所指的记录太小时，导致后插入的记录永远也不能比它小，使得被删除的链表空闲空间很多，却一直用不到的问题，其处理方式是，如果这个页面最后实在没有空间存储新记录，那么此时系统会做一次页面内的全面重组，重组方式及所要做的工作在第 8 章中已经介绍过了。

上面讲了 **heapno**，貌似扯远了，其实不然，因为它是产生 **GAP** 锁的源头，如果想要更深层地理解或了解 **GAP** 锁，就必须首先要理解 **heapno**。

下面再回过头来看死锁问题。

在表为空表的情况下，事务 2 插入的 (8, 6) 在唯一索引对应的页面中。因为此时还是空表，所以这条记录得到的 **heapno** 应该是 2。这里只谈论这个页面，所以前提是假设在这个页面中，它已经插入完成。

在事务 1 中插入 (6, 6) 时，因为 **name** 上面的索引是唯一索引，所以需要检查唯一性，它上的锁是 **S** 锁。在检索之后，发现已经存在这样一条记录与它要插入的是冲突的，而且这个记录是别的事务插入的，这个事务还处于 **ACTIVE** 状态，那么事务 1 就会先“帮忙”给这个

记录上一个写锁。这里需要注意的是,事务 1 给事务 2 上锁,其实是将一个隐式的锁转换为显式的锁,锁类型为 LOCK (X|NOTGAP|REC)。然后再给自己上锁,因为这个记录上面已经有了 X 锁,自己再上 S 锁肯定冲突,所以加锁之后就是 LOCK (S|WAITING),处于等待状态。

然后,事务 2 再插入一条记录 (7, 5),此时关键的问题出现了,在唯一索引中要插入的是 5,那么系统首先会找一个逻辑上合适的位置,因为它是排序的,所以逻辑上要插入到 (8, 6) 前面, supremum 后面。这种情况,系统的实际定位是 supremum 记录的位置,然后将新记录插入到后面。

在当前配置下, InnoDB 插入记录采用的是 GAP 锁,策略是 next-key-lock,当前所定位到的插入位置为 supremum 记录之后,而之后的一条记录为之前插入的第一条,其 heapno 为 2,插入线程会判断当前 heapno 为 2 有没有锁。而此时显然已经加了两个锁,分别为 #1 (@1 语句加的锁): LOCK (X|NOTGAP|REC)、#2 (@2 语句加的锁): LOCK (S|WAITING)。在检查锁冲突时,要与当前记录上最近加的锁做对比,此时找到的锁为 #2,因为已经有锁了,所以当前插入事务要新加的锁为 #3: LOCK (LOCK_X|LOCK_GAP|LOCK_INSERT_INTENTION),且由于锁是 GAP 类型,同时是插入的,所以就是这三个的组合锁。此时,就要判断 #2 和 #3 是不是冲突了。很明显,这两个锁是冲突的,一个是读一个是写。

判断之后,会做一次死锁检查扫描,锁 #3 等待锁 #2,锁 #2 此时正在等锁 #1,而锁 #3 和锁 #1 所对应的事务都是事务 2,这里出现了环和死锁。

死锁出现之后,如何解锁?在数据库中,一般采用杀掉某一个事务的策略,那么在 InnoDB 中,又是如何选择应该杀掉哪一个的呢?

继续上面的例子,锁 #3 等待锁 #2,锁 #2 发现它等待的锁为 #1,而此时 #1 正好又与 #3 是同一个事务,锁 #2 是处于等待状态,锁 #1 处于被等待状态,系统此时采取的方式是将锁 #2 对应的事务杀掉,也就是回滚掉,这也证明了前面 show engine innodb status 中显示的是将事务 1 回滚掉的事实。

下面考虑另外三个问题。

- 上面的例子中,如果事务 2 第 2 次插入的数据是 (9, 7),死锁会不会出现?对于这个问题,答案是否定的。单看这个场景,插入 7 的时候,系统定位的位置肯定是刚插入的值为 6 的记录,使用的是 next-key-lock,它会判断 6 后面的记录有没有上锁,此时发现没有上锁,则 7 这个记录顺利插入。
- 如果表不是空表,会不会有问题?对于这个问题,答案是肯定的。因为这个问题与数据库表中是不是有数据没有关系。假设这个表中有 100 万条数据,对应地要插入页面中 N 条记录,事务 2 在某一个位置插入一条记录之后,同样产生一个 heapno。如果事务 1 插入同样一条记录,此时得到的 heapno 肯定还是一样的,事务 1 会继续等。如果在事务 2

中再插入一条比刚才插入的小的记录，这条记录正好落在了上锁记录与这个记录的前一个记录之间，那么就会出现死锁现象。

- 如果 name 列的索引不是唯一索引，会出现死锁么？对于这个问题，答案是否定的。因为如果不是唯一索引，事务 1 根本不会去检查二级索引的唯一性，不会上 S 锁，当然也不会将隐式的锁改为显式的，所以会直接插入完成。因为没有唯一性，索引值是相等的（都是 6），这种情况是直接将记录追加到后面去，所以这个操作就直接完成了，不会阻塞，更不会死锁。

发散思维

上面的例子是，唯一索引只有一个列，如果这个唯一索引有多个列，那么当插入数据的前缀有若干个列相同的时候，在同样操作的情况下，也会出现死锁，而当只有中间的某几个列相同的时候，则不会出现这个问题。

总结

在 MySQL 数据库中的死锁，如开头所说，是非常常见的事情，总结一下本案例的死锁原因，有如下三点。

- 这个死锁问题最主要的原因就是中间出现了对唯一索引的唯一性检查。唯一性检查时，将一个隐式的锁改为显式的，将本来不会出现阻塞的插入操作阻塞了，最终导致了这个死锁的出现。
- 使用了 REPEATABLE-READ 的隔离级别。在这种隔离级别下，InnoDB 为了防止产生幻读现象，使用了 GAP 锁的方式来处理插入操作，导致锁的粒度可以变得非常大，从而进一步导致了死锁的频繁出现。所以如果没有特别需求，MySQL 的默认隔离级别可以首选设置为 READ-COMMITTED 级别，这样可以避免很多不必要的问题。
- GAP 锁也是一方面，不过这与第二点是关联的。

19

MySQL 在并发删除同一行数据时 导致死锁的分析

背景

死锁，对于 MySQL 来说，是一件习以为常的事情。有很多原因导致了 MySQL 的死锁率很高，包括隔离级别、GAP 锁、索引的顺序及业务逻辑等，所以分析死锁，可能会非常困难。因为 MySQL 关于死锁提供的日志信息有限，并且不完善，所以很难重现死锁发生时的场景。但有一些比较常见的死锁，还是有迹可寻，所以本章就通过一个死锁的案例，根据各种资源来追本溯源，找到死锁发生的真正原因，并且希望通过其中一些方法，可以起到举一反三的作用。

前面已经介绍了关于二级唯一索引死锁的例子，现在再介绍另一个关于删除操作引起的死锁问题。

有一个朋友问我，为什么多个线程同时做删除同一行数据的操作，总是报死锁，线上已经出现了好多次，我问了他如下三个问题。

- 是不是在一个事务中做了好几件事情？答：不是，只做一个删除操作，自动提交。
- 有多少个线程在做删除操作？答：差不多 10 个。
- 是什么隔离级别？答：可重复读的隔离级别。

当时觉得不可思议，按说对于行锁而言，如果已经有事务加锁了，则自动提交会等待。等提交之后，发现记录已经被删除了，就会返回，删除 0 条，为什么会死锁？

然后，写一个程序，模拟线上的并发，死锁就可以重现，但不一定是必然重现，只是偶然性地出现几次。所以，这种问题比较难找到原因。不过，只要能重现，就有找到真正原因的希望。

表结构如下。

```
CREATE TABLE `abcdefg` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `abc` varchar(30),
  `def` varchar(30) ,
  `ghi` date,
  `jkl` date,
  `mnp` tinyint(4),
  PRIMARY KEY (`id`),
  UNIQUE KEY `uniqudefghijkl` (`def`,`ghi`,`jkl`)
);
```

运行环境如下。

- 这个表包括两个索引，一个是聚簇索引，另一个是 `uniqudefghijkl` 的二级唯一索引。
- 事先插入很多数据，然后三个线程同时删除同一条记录。这里只做删除操作，并且是自动提交，为了得到一批要删除的数据，事先查询出很多条来备用。
- 隔离级别是 `REPEATABLE-READ`。

删除语句如下。

```
delete from abcdefg WHERE abc= '我是变量' and def= '我是变量'
and ghi= '2013-12-19 00:00:00' and jkl= '2013-12-20 00:00:00';
```

问题分析

运行并发程序，死锁果然很快就出现了，下面是执行 `SHOW ENGINE INNODB STATUS` 的结果。

```
LATEST DETECTED DEADLOCK
-----
140123 12:20:50
*** (1) TRANSACTION:
TRANSACTION 2E10, ACTIVE 4917 sec starting index read
MySQL tables in use 1, locked 1
```

```

LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s)
MySQL thread id 3, OS thread handle 0x1008, query id 43 192.168.xx.x username updating
delete from abcdefg WHERE abc= '我是变量' and def= '我是变量' and ghi= '2013-12-19 00:00:00' and jkl= '2013-12-20 00:00:00';
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 12295 n bits 528 index `uniqdefghijkl` of table `deadlock`.`abcdefg` trx id 2E10 lock_mode X locks rec but not gap waiting
Record lock, heap no 167 PHYSICAL RECORD: n_fields 4; compact format;

*** (2) TRANSACTION:
TRANSACTION 2E0E, ACTIVE 4917 sec starting index read
MySQL tables in use 1, locked 1
3 lock struct(s), heap size 1248, 2 row lock(s)
MySQL thread id 1, OS thread handle 0x1190, query id 41 192.168.xx.xx username updating
delete from abcdefg WHERE abc= '我是变量' and def= '我是变量' and ghi= '2013-12-19 00:00:00' and jkl= '2013-12-20 00:00:00';
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 0 page no 12295 n bits 528 index `uniqdefghijkl` of table `deadlock`.`abcdefg` trx id 2E0E lock_mode X locks rec but not gap
Record lock, heap no 167 PHYSICAL RECORD: n_fields 4; compact format;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 12295 n bits 528 index `uniqdefghijkl` of table `deadlock`.`abcdefg` trx id 2E0E lock_mode X waiting
Record lock, heap no 167 PHYSICAL RECORD: n_fields 4; compact format;
*** WE ROLL BACK TRANSACTION (1)

```

经过测试,发现在三个线程的情况下可以重现死锁,但在2个线程的情况下死锁不能重现,这样无疑将问题复杂化了。因为在上面的日志信息中,只能看到两个事务的信息,没有第三个事务的信息,而这个死锁是需要三个或以上事务才会出现,这很明显如果只是从日志信息中找问题原因,必定是无能为力的。

这下坏了,多线程调试很麻烦,有时候这个线程运行了,另一个又不运行了,如果暂停了某个线程,有可能导致线程之间死锁,而如果自然执行,就又不能出现死锁的情况。因为这个死锁也是偶然性的,所以最终只有一种方法,那就是在MySQL代码中打印log信息,将锁、记录与事务这块的函数中具有分歧点的地方都加上注释,并且将有用的信息打印出来,最终分析log文件,才能发现死锁的真正猫腻。

通过分析日志文件，知道了上面死锁产生的原因，知道了哪个事务做了什么事导致了另一个事务的等待，等等之类的信息。最后，总结了这个死锁案例的事务时序图，如下表所示。

事务 A	事务 B	事务 C
开始		
表的 IX 锁 17@1		
二级索引行锁 X REC NOTGAP 1059 @2 检查死锁没事		
	表 IX 锁 17@3	
	二级索引记录行锁 REC NOT- GAP X WAIT 1315 @4 检查死 锁，没事	
		表 IX 锁 17@5
		二级索引记录行锁 REC NOT- GAP X WAIT 1315 @6 检查死锁 没事
聚簇索引行锁 X REC NOTGAP 1059 @7		
	wait....suspend....	wait....suspend....
commit		
	wakeup this trx 将 @4 的 WAIT 去掉，成为 1059	
	二级索引记录行锁 REC X WAIT 291 @8 检查死锁发现死 锁	

上面的数字都是源代码中关于各种锁的位图值，说明如下。

- LOCK_TABLE: 16。
- LOCK_IX: 1。
- LOCK_REC_NOT_GAP: 1024。
- LOCK_WAIT: 256。
- LOCK_REC: 32。
- LOCK_X: 3。

所以, 锁 @6 表示的是 LOCK_REC & LOCK_REC_NOT_GAP & LOCK_X & LOCK_WAIT = 1315, 以此类推。

这里检查死锁的算法大体上就是检查有没有形成等待环。

而这个案例的死锁环表现为: 事务 B 的锁 @8 等待事务 C 的锁 @6, 事务 C 的锁 @6 在等待事务 B 的锁 @3, 此时发现又绕回来了, 产生死锁。

到这里, 死锁的现象如何产生已经解释清楚了。但是, 为什么会这样呢? 这里的疑问是: 在事务 A 提交之后, 事务 B 被唤醒, 事务 B 的锁 @4 为 REC NOTGAP X (1059), 但由于被唤醒之后, 环境变了, 所以需要重新上锁, 上锁之前, 会检查当前事务是否对这条记录已经加了兼容的锁, 如果有则直接使用并继续执行, 如果没有则再加锁, 做这个检查的函数为 lock_rec_has_expl, 它做的检查如下。

```
UNIV_INLINE
lock_t*
lock_rec_has_expl(
    uint         precise_mode, /*!< in: LOCK_S or LOCK_X
                               possibly ORed to LOCK_GAP or
                               LOCK_REC_NOT_GAP, for a
                               supremum record we regard this
                               always a gap type request */
    const buf_block_t* block, /*!< in: buffer block containing
                               the record */
    uint         heap_no, /*!< in: heap number of the record */
    trx_id_t     trx_id) /*!< in: transaction id */
{
    /* local variables ... */
    lock = lock_rec_get_first(block, heap_no);
    while (lock) {
        /* 首先遍历位置heap_no对应记录上所有的锁, 找到属于当前事务的锁 */
        if (lock->trx == trx
            /* 要确定已经加上的锁不是处于等待状态 */
            && !lock_is_wait_not_by_other(lock->type_mode)
            /* 要确定已经加上的锁的类型的兼容性, 比如XX、SS、XS等是否冲突。
             这里返回真的条件是已经存在的锁至少要和新加锁是相同级别的。在这里,
             因为要加的是X锁, 所以已经存在的锁至少是X模式的才能返回真 */
            && lock_mode_stronger_or_eq(lock_get_mode(lock),
                                       precise_mode & LOCK_MODE_MASK)
        /* 这里是用来判断锁类型。如果其类型包含LOCK_REC_NOT_GAP, 或者
         已经存在的锁不是LOCK_REC_NOT_GAP类型的, 则可以返回真 */
        )
            return lock;
        lock = lock_rec_get_next(lock);
    }
    return 0;
}
```

```

    && (!lock_rec_get_rec_not_gap(lock)
        || (precise_mode & LOCK_REC_NOT_GAP)
        || heap_no == PAGE_HEAP_NO_SUPREMUM)
    /* 已经存在的锁不是GAP类型的，或者新加的锁为GAP类型，就返回真 */
    && (!lock_rec_get_gap(lock)
        || (precise_mode & LOCK_GAP)
        || heap_no == PAGE_HEAP_NO_SUPREMUM)
    && (!lock_rec_get_insert_intention(lock)))
{
    /* 如果上述条件结果为真，则说明找到了当前事务已经存在的锁，
    直接使用这个锁，不会再新加锁了 */
    return(lock);
}

lock = lock_rec_get_next(heap_no, lock);
}
}

```

针对当前事务已经存在的锁，以及打算新加的锁，需要满足以下六个条件。

- 首先这个锁是自己事务产生的。
- 已经存在的锁不是处于等待状态。
- 当前锁的类型与要新加锁的类型 precise_mode 是兼容的，precise_mode 值是 X 锁，因为这里要做删除。
- 已经存在的锁不是 NOT GAP 类型，或者打算新加的锁类型是 NOT GAP 类型的。
- 已经存在的锁不是 GAP 类型，或者要加的锁类型是 GAP 类型的。
- 当前锁不是意向插入锁。

当前事务在加锁之前，做检查调函数 lock_rec_has_expl 时，已经存在的锁为 1059 (锁 @4，去掉了 WAIT 属性，还是 NOT GAP 类型的)，同时新加的锁类型不是 NOT GAP 的，与第 4 点正好相反，整个条件判断结果为假，所以没有找到，此时就需要在外面重新创建一个锁。因为此时这行已经有锁了，那么它会创建一个 REC WAIT X 锁 (291)，也就是锁 @8。

所以，即使锁 @4 不是处于等待状态了，此时也不能直接执行，而是重新创建一个锁。新创建的锁 @8 要等待锁 @6，而 @6 此时在等待 @4，@4 和 @8 是同一个事务的，所以此时导致了死锁。

那么现在问题又来了，从上表可以看到，这个时间序列没有什么特别的，或者通过构造一个类似的交叉过程，来测试一下是不是可以很容易重现呢？

测试方法是通过开启三个会话的方式，将其都设置为 not autocommit，这样才能让事务 A 在事务 B 和事务 C 开始之后提交，来保证与上面表格中类似的时序关系。

那么接下来，开始创建相同的表，删除同一行记录，如下表所示。

事务 A	事务 B	事务 C
begin		
delete, 删除行数为 1		
	begin	
	delete 阻塞	
		begin
		阻塞
commit		
	观察有没有死锁，其实并没有，删除行数为 0	
		删除行数为 0

表面上看，这两张表没有什么区别，但第 2 张表中没有显示出现死锁。这是为什么？

其实，没有死锁是正常的，如果这样就死锁了，那 MySQL 就简直不能用了！

看来还是有区别的，正常模式下再做一次 log 分析，从 log 中看出了大问题。

上面详细的加锁表在无死锁情况下（因为死锁是偶然出现的）对应的时序表如下。

事务 A	事务 B	事务 C
开始		
表的 IX 锁 17 @1		
二级索引行锁 X REC NOTGAP		
1059 @2 检查死锁没事		
聚簇索引行锁 X REC NOTGAP		
1059 @7 检查死锁没事		
	表 IX 锁 17 @3	
	二级索引记录行锁 REC X	
	WAIT 291 @4 检查死锁，没事	
		表 IX 锁 17 @5

续表

事务 A	事务 B	事务 C
		二级索引记录行锁 REC X WAIT 291 @6 检查死锁没事
	wait.... suspend....	wait.... suspend....
commit		
	wakeup this trx 将 @4 的 WAIT 去掉, 成为 35	
	执行完成, 提交	
		执行完成

此时发现, 上面这张表其实与第 1 张表是一样的, 那么为什么上面这张表显示可以正常执行完成, 而第一张表却出现死锁了呢?

认真仔细看了之后, 发现有很小的地方是不同的, 图 3 中的锁 @4 加上的锁是 291 (REC & X & WAIT), 而第 1 张表中加的锁比它多了一个 NOTGAP 的锁, 锁 @6 也是一样的, 第 3 张表的事务 A 在提交并且唤醒了锁 @4 之后, 它的锁类型为 REC+X (35), 而第一张表中的值也比它多了一个 NOTGAP 锁。

现在, 已经基本定位了问题所在, 应该是 NOTGAP 搞的鬼。但是为什么会有差别呢?

此时, 还需要回到代码中查看, 通过日志分析, 发现上面两种情况在执行下面代码时走了不同的路 (截取 row_search_for_mysql 函数中的一段代码), 如下。

```
if (prebuilt->select_lock_type != LOCK_NONE) {
    uint lock_type;
    if (!set_also_gap_locks
        || srv_locks_unsafe_for_binlog
        || trx->isolation_level <= TRX_ISO_READ_COMMITTED
        || (unique_search && !UNIV_UNLIKELY(rec_get_deleted_flag(rec, comp)))) {
        /* 如果满足上面的条件, 就会直接跳到下面 “lock_type = LOCK_REC_NOT_GAP;” 的
           位置, 条件基本包括两个条件:
           1. 隔离级别在 READ_COMMITTED 及以下
           2. 当前记录没有打删除标志, rec_get_deleted_flag 函数的作用是判断这条
              记录是不是已经打了删除标志 */
        goto no_gap_lock;
    } else {
        lock_type = LOCK_ORDINARY;
    }
}
```

```

    }

    if (index == clust_index
        && mode == PAGE_CUR_GE
        && direction == 0
        && dtuple_get_n_fields_cmp(search_tuple)
        == dict_index_get_n_unique(index)
        && 0 == cmp_dtuple_rec(search_tuple, rec, offsets)) {
no_gap_lock://标记
        lock_type = LOCK_REC_NOT_GAP;
    }


```

现在豁然开朗了, 如果当前这条要加锁的记录没有打上删除标志, 则加的锁是 NOTGAP 类型的锁, 否则就不设置类型。那么说明上面第一张表中的事务 A 还是有一个细节没有画出来, 正因为这个细节与事务 B 发生了交叉, 导致了事务 B 在做的时候还没有打上删除标志, 所以就加了 NOTGAP 锁, 而在被唤醒之后, 再加锁时, 因为已经打了删除标志, 导致加锁类型为 REC X, 进而导致前面加的锁与后面加的锁不匹配, 所以导致了后面的死锁。

从第 2 张表所展示的测试来看, 因为事务 A 已经完成了所有的操作, 只等待提交, 此时肯定已经打了删除标志, 在事务 B 和 C 操作时, 他们从前到后加的锁都是 LOCK_ORDINARY 锁, 所以就不会出现死锁。

现在已经确定, 问题就出现在上面代码的判断中, 在上面代码的上面还有一段注释, 如下。

```

 /* Try to place a lock on the index record; note that delete
marked records are a special case in a unique search. If there
is a non-delete marked record, then it is enough to lock its
existence with LOCK_REC_NOT_GAP. */

```

这说明了加 NOTGAP 锁的意图, 即上面代码的判断是专门做的, 具体原因就无从查起了, 但是注释中说这是一种特殊情况, 为什么呢? 解决方式是把那 2 行直接去掉就可以了 (测试过不会出现死锁了), 但这是否是解决问题的根本原因, 还要等待官方人员的处理。

所以到这里, 完整的死锁表如下所示。

事务 A	事务 B	事务 C
开始		
表的 IX 锁 17@1		
二级索引行锁 X REC NOTGAP		
1059 @2 检查死锁没事		
	表 IX 锁 17@3	

续表

事务 A	事务 B	事务 C
	二级索引记录行锁 REC NOT-GAP X WAIT 1315 @4 检查死锁，没事	
		表 IX 锁 17@5
		二级索引记录行锁 REC NOT-GAP X WAIT 1315 @6 检查死锁 没事
对二级索引记录加删除标志这个是最关键的这个关键点必须要出现在锁 @4 与 @8 之间		
聚簇索引行锁 X REC NOTGAP 1059 @7		
	wait....suspend....	wait....suspend....
commit		
	wakeup this trx 将 @4 的 WAIT 去掉变成 1059	
	二级索引记录行锁 REC X WAIT 291 @8 检查死锁发现死锁	

其实到这里，已经解释清楚了死锁与不死锁两种情况的分歧点在哪里，以及导致这样分歧的原因。分歧点就是事务 A 对二级索引打删除标志必须要出现在事务 B 所上的锁 @4 之后，同时要出现在事务 B 上锁 @8 之前，这样就出现了一个事务对同一行记录前后上锁类型不一致的情况，从而导致了死锁。

此时可能有人要问，事务 B 在被唤醒之后，为什么还要再上锁？有了锁 @4 还不够么？这是因为在事务 B 被唤醒之后，记录状态有可能会改变，而此时确实改变了，记录已经被打上了删除标志，参考上面的代码，上锁不会是 LOCK_REC_NOT_GAP 类型的，所以需要加的锁为 291（REC X，WAIT）。在上锁之前，通过 lock_rec_has_expl 来检查时发现，已经存在的 @4 与要新加的锁不匹配，不能直接使用，这就回到了之前讲函数 lock_rec_has_expl 的内容了。所以，需要新加锁，就是锁 @8，此时再次去检查是不是死锁，发现死锁出现了，@8 等 @6，@6 等 @4，而 @4 和 @8 是同一个事务，所以死锁了。

发散思维

可能有人通过上面的代码看到 (函数 `row_search_for_mysql` 的实现), 判断条件有一个是 `trx->isolation_level <= TRX_ISO_READ_COMMITTED`, 也就是说, 如果隔离级别为 `READ_COMMITTED` 时, 不管有没有打删除标志, 都会加上 `LOCK_REC_NOT_GAP` 锁, 那这样不也会死锁?

其实不是这样的。这里死锁的根本原因是, 事务 B 在被唤醒之后, 想要加的锁与已经存在的锁类型不匹配, 才需要新加一个锁。而不匹配的原因是在两个锁之间, 对应的记录被事务 A 打了删除标志, 从而在第二次加锁时, 导致条件 `!UNIV_UNLIKELY(rec_get_deleted_flag(rec, comp))` 的值为假, 进而加了 `LOCK_ORDINARY` 的锁, 导致与第一次不匹配。

而如果隔离级别是 `READ_COMMITTED` 的情况, 就与上面是不是打了删除标志的条件没有关系了, 它只会考虑隔离级别的条件, 而这个条件, 永远为真, 也就是说, 不管在何种情况下, 加的锁都会是 `LOCK_REC_NOT_GAP` 类型的。在事务 B 被唤醒之后, 试图加锁并通过函数 `lock_rec_has_expl` 来判断时, 发现当前事务已经有一个同样的锁了, 所以计划要加的锁就不需要了, 直接使用锁 @4 即可, 从而不会出现环形等待及死锁的现象了。

总结

在 MySQL 中, 其实很多东西都不能按照常理来想, 这个问题本来在达梦与 Oracle 中是根本不可想象的, 也根本不会出现。所以, 一开始才觉得不可能, 最后才发现, 原来是真的。

针对这个问题, 有如下四点总结。

- 与之前所说的情况一样, 此次死锁还是与事务的隔离级别设置为 `REPEATABLE-READ` 有关。这是一大害啊! 所以, 如果可以, 或者新业务上线的话, 要将默认设置设置为 `READ_COMMITTED` 隔离级别。现在应该知道, 将隔离级别设置为 `READ_COMMITTED` 真是解决死锁的一大法宝!
- 这个死锁现象, 必须要有三个事务一起才会出现, 所以单从 `show engine innodb status\G` 信息中根本查不到这个问题的原因。
- 死锁的偶然性, 是由于第 4 张表中所介绍的“关键点”导致的。这个死锁的发生, 必须要按照前面表格所列出来的时序在 InnoDB 存储引擎内部运行, 否则死锁不会出现。关键问题是, 要保证事务 B 和事务 C 前面获取行锁时, 这行还没有被打上删除标志, 然后再去等待, 而之后, 事务 A 才去删除这条记录, 打上标志, 然后事务 A 提交, 从而在事务 B 重启事务之后, 前后记录状态不一致导致重新加锁, 进而导致了死锁的发生。而这个顺序就是偶然性的。
- 这种问题的死锁解决方法, 就是要从业务逻辑上解决, 可以直接将并发数目限制为两个, 便不会再出现死锁, 或者干脆不要有相同数据的大量并发删除现象, 又或者将隔离级别设置为 `READ_COMMITTED` 级别。

参数 SQL_SLAVE_SKIP_COUNTER 的 奥秘

每次数据库复制冲突之后，经常使用的一个命令如下。

```
 SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
```

一般会认为，现在出现冲突错误，那就将上面参数值设置为 1，跳过出错的这个 event 就可以解决了。重新启动复制，发现问题果然解决，我们以为这样理解是正确的。

其实不然。

这需要从 Binlog 的内容说起。在 Binlog 中，所有的 Binlog 是按照组来分的，每一个组是其主库生成的一个事务 Binlog，都以 BEGIN 开始并以 COMMIT 结束。还有一些特殊事件，比如用户变量的设置、随机数的设置等。

那么，设置参数 SQL_SLAVE_SKIP_COUNTER 之后，对复制的影响是什么呢？从库遇到这个参数时，它的 skip 算法又是什么呢？

这需要根据不同类型的事件，分别做介绍，如下。

- BEGIN 语句：对于一个 Binlog 组，肯定会有一个 BEGIN 语句作为开始的标志。执行到 BEGIN 时，说明从这个位置开始，到后面出现的第一个提交事件结束，中间这一段 Binlog 是属于一个组的，那么此时会因为不同的 SQL_SLAVE_SKIP_COUNTER 有不同的处理方式。如果参数 SQL_SLAVE_SKIP_COUNTER 值为 1，则此时这个组中的所有事

件都会被算作不计数事件，也就是说这个 1 代表一个事务，而不是一个事件。只有遇到 COMMIT 语句时，才会将计数 1 减为 0，那么下一个事务组会正常执行。如果参数 SQL_SLAVE_SKIP_COUNTER 的值大于 1，那么这个组中的事件就会被认为是一个个的事件，处理一个事件，参数 SQL_SLAVE_SKIP_COUNTER 的值就减去 1。当减到 1 的时候，如果这个事务组还没有结束，则回到上面，跳过值为 1 的情况；如果结束了，但还没有减为 0，那么下一个事务组会继续重新处理。

- COMMIT 或 ROLLBACK 语句：上面已经提到，遇到这个语句时，不管任何情况，参数 SQL_SLAVE_SKIP_COUNTER 的值都会减 1。如果 SQL_SLAVE_SKIP_COUNTER 的值为 1，就像上面说的，COMMIT 之前所有的事件都会被算为不计数事件，这里减 1 之后，就完成了了一个事务的 skip。
- 其他 Query 语句：上面已经说过，当 SQL_SLAVE_SKIP_COUNTER 为 1 的时候，会将组内事件都跳过，否则它会被减 1。
- Rows 事件：这种类型是在行模式下，一个行的事件类型。针对一条 sql 语句产生的若干个事件，分别计数。如果设置的 SQL_SLAVE_SKIP_COUNTER 大于 1，则针对每一个事件都会递减 1，如果减到了 1 或设置为 1 了，则直到 COMMIT 事件之后才会计数，之前所有的操作都不会被计数。
- 不计数事件：这种类型的意思是，只要遇到这种事件，并且设置了参数 SQL_SLAVE_SKIP_COUNTER 为 1 或递减之后值为 1，那么就跳过，并且不会影响 SQL_SLAVE_SKIP_COUNTER 的值。而如果设置的 SQL_SLAVE_SKIP_COUNTER 值大于 1，则计数递减 1，直到递减到 1 时这类事件才不会算入计数。这种类型的事件包括 Table_map、Intvar、Rand、User_var、BEGIN_load 这五个事件，所以在 Binlog 中如果有这五个事件，则在计数中不做计数，直接忽略。

需要注意的是，在每次复制中断后，看上去中断的位置是出错的事件，但实际上，那只是一个执行错误的位置。因为在复制时是以组（事务）为单位的，事务中执行出错了，则这个事务会回滚，这个组就没有完成。实际上，此时 Binlog 停止的位置是这个组的开始位置，所以在设置好之后，开始的位置肯定是 BEGIN 或 BEGIN 之前的一些设置命令的位置，此时设置 SQL_SLAVE_SKIP_COUNTER 为 1 之后，后面跳过的就是一个完整的事务，而不是一个事件而已。

对于设置 SQL_SLAVE_SKIP_COUNTER 为其他值的情况，这是比较危险的。因为它的跳过算法在上面已经讲清楚了，它会把每一个 query 语句（包括 BEGIN 及 COMMIT）都计入跳过计数的情况，也就是说，假设一个组至少存在 3 个事件，那么如果设置跳过为 4、5、6，实际上结果有可能只跳过 1 个事务，也有可能跳过 2 个事务，但这是没有办法预期的。除非你自己已经很清楚当前点之后有多少个事件及对应事件的类型，否则不会知道具体跳过了几个事务。

此时可以再回到开头所说的问题上来。在 skip 的时候，看到的是在哪个位置出错了，但实际上此时是停在了一个事务的开始位置，而出错的位置有可能是在事务中的某一个语句或者行上面，那么此时做 skip，实际上是跳过了当前中断位置所在的整个事务。可想而知，事务中如果有其他操作，也就都跳过了。而从表象上看，好像是跳过了这个事件。所以说，看到的和真实发生的其实不太一样。

当然通过设置参数 SQL_SLAVE_SKIP_COUNTER 来跳过复制错误的操作，只有在 MySQL 5.5 版本，或者是 5.6 及以上的版本中没有开 GTID 的情况下才能使用。在处理问题时，有时候确实很方便，但也是比较危险的，最好确认清楚是不是可以跳过，跳过之后，是不是要处理数据丢失的问题等。最好在跳过之前，记录一下相关 Binlog 的位置，在恢复之后，看一下从出错位置开始的一个 Binlog 事务，有没有需要特殊处理的操作。

关于这个问题，其实很容易做一些测试，研究一下参数 SQL_SLAVE_SKIP_COUNTER 设置为不同的值时，复制是什么表现。不过，个人建议永远不要将这个参数的值设置为非 1，否则会非常难控制。假设跳过的值太多，可以分开多次，每次跳过最多一个事务，这样也能做到心中有数，并且是只有出错的时候才去跳。

下面是一段每次跳过一个事务的脚本，只有在复制中断的情况下才会跳过，并且自动开始复制。



```
#!/usr/bin
MySQL_user=username
MySQL_password=password
MySQL_host=127.0.0.1
MySQL_port=3306
sleep_interval=100

while :
do
    date
    mysql -u${MySQL_user} -p${MySQL_password} -h ${MySQL_host} -P ${MySQL_port} -e
    "set global sql_slave_skip_counter=1;start slave;"
    usleep ${sleep_interval}
    echo
done
```

当然，可以对这段代码稍微做一点改造，加上一行可以记录一下中断时的位置。每跳过一个事务，都打印一下这个事务的开始位置，这样可以了解跳过的事务量，并且方便事后查找跳过了哪些事务。

但这样大批量的跳过，一般是在处理故障或是明知道影响不大时才这样做的。还是那句话，请谨慎使用。

21

Binlog 中的时间戳

背景

众所周知，在 Binlog 文件中，经常会看到关于事件的时间属性，出现的方式都是如下这样的。

```
#161213 10:11:35 server id 11766  end_log_pos 263690453 CRC32 0xbec3aaf5 Xid =  
83631678
```

我们清楚地知道，161213 10:11:35表示的就是时间值，但除此之外呢？还能知道它的什么信息呢？

问题分析

先从一个典型的案例入手来讲述其中的细节，比如曾经在 Galera Cluster 碰到的一个问题，可以先看一段 Binlog 内容，如下。

```
#161213 10:11:35 server id 11766  end_log_pos 263677208 CRC32 0xbfc41688  
GTID [commit=yes]  
#161213 10:10:44 server id 11766  end_log_pos 263677291 CRC32 0x02537685  
Query  thread_id=4901481  exec_time=0 error_code=0  
#161213 10:10:44 server id 11766  end_log_pos 263677435 CRC32 0x0e70aab6  
Write_rows: table id 17852 flags: STMT_END_F
```

```

#161213 10:10:44 server id 11766 end_log_pos 263677609 CRC32 0xb58d4c61
  Update_rows: table id 17853 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263690453 CRC32 0xbec3aaf5
  Xid = 83631678
#161213 10:11:30 server id 11766 end_log_pos 263690501 CRC32 0x6e798470
  GTID [commit=yes]
#161213 10:11:30 server id 11766 end_log_pos 263690592 CRC32 0x2b6a6d34
  Query thread_id=4900813 exec_time=5 error_code=0
#161213 10:11:30 server id 11766 end_log_pos 263691291 CRC32 0xc0f9ed87
  Update_rows: table id 17891 flags: STMT_END_F
#161213 10:11:30 server id 11766 end_log_pos 263691322 CRC32 0xe40764c4
  Xid = 83631679
#161213 10:11:35 server id 11766 end_log_pos 263691370 CRC32 0xbaa4ca30
  GTID [commit=yes]
#161213 10:11:35 server id 11766 end_log_pos 263691453 CRC32 0x030f321c
  Query thread_id=4900813 exec_time=0 error_code=0
#161213 10:11:35 server id 11766 end_log_pos 263692240 CRC32 0x7584d6a1
  Delete_rows: table id 73 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263692271 CRC32 0x03abb120
  Xid = 83631680

```

上面列出来的是被处理过的 Binlog 内容，其中包括三个事务，每个事务只列出了标志性的事件，比如事务开始的 GTID 事件、执行线程信息及提交事件等。出现的顺序，就是 Binlog 内容的顺序，这一点可以从 Xid 的连续性看出来。

在上面一段内容中，重点关注一下时间信息。每一个事务中的每一个事件都有时间属性，可以看到，第一个事务是在 10:11:35 时间点提交的，第三个事务也是在这个时间提交的。但中间一个事务，即 Xid 为 83631679 的事务，包括提交时间在内的所有事件时间，都是在 10:11:30 时间点发生的，比其他两个事务足足早出现了 5 秒钟！

很多同学看到这样的现象之后，可能会有以下的疑问。

- 在 Binlog 文件中，不是以提交顺序存储的么，前面提交的事务怎么会存储在后面的位置呢？
- 假设事务 83631679 的开始时间是 10:11:30，那么至少它的提交事件，即 GTID 事件的时间是 10:11:35 吧？
- 事务 83631679 的执行时间是 5 秒钟，从 exec_time=5 可以看出来这个信息的出现，那么第二个问题就变得更加让人疑惑了。

这里能看出 exec_time=5 这样的信息，值得称赞，这是一个很重要的信息。因为现在明确地知道，事务 83631679 是在 10:11:30 开始的。那么，这个 Query 又执行了 5 秒钟，可以和事

务提交时间 10:11:35 对得上。现在要明确的一点就是，事务是在 10:11:35 提交的，只不过在 Binlog 内容看到的是 10:11:30，那就要弄清楚 Binlog 在记录时间戳的问题上，是如何处理的。时间戳是一个事件的属性，但这个属性的来源是哪里，也就是说这个时间是什么时候记录下来的，可以看如下一段代码。

```
Log_event::Log_event(
    THD* thd_arg, uint16 flags_arg,
    enum_event_cache_type cache_type_arg,
    enum_event_logging_type logging_type_arg,
    Log_event_header *header, Log_event_footer *footer)
: is_valid_param(false), temp_buf(0), exec_time(0),
  event_cache_type(cache_type_arg), event_logging_type(logging_type_arg),
  crc(0), common_header(header), common_footer/footer, thd(thd_arg)
{
    server_id= thd->server_id;
    common_header->unmasked_server_id= server_id;

    /* 这里就是用来在创建一个事件时，给这个事件的时间赋值的过程。
       可以看到，这个时间的来源是thd->start_time的值，那我们需要
       做的，就是弄清楚这个值的来源了 */
    common_header->when= thd->start_time;
    common_header->log_pos= 0;
    common_header->flags= flags_arg;
}
```

如代码中的解释，需要找到 `thd->start_time` 的来源。这个值的设置很容易就可以找到，在每一条语句执行前都会做一次，通过函数 `thd->set_time()` 来设置。其中一个很重要的 MySQL 语句，在入口处理函数中就调用了，可以简单看一下，如下。

```
bool dispatch_command(THD *thd, const COM_DATA *com_data,
                      enum enum_server_command command)
{
    /* local variables... */

    thd->set_command(command);
    /*
       Commands which always take a long time are logged into
       the slow log only if opt_log_slow_admin_statements is set.
    */
    thd->enable_slow_log= TRUE;
    thd->lex->sql_command= SQLCOM_END; /* to avoid confusing VIEW detectors */
}
```

```

/* 就是在这里，初始化这个时间为当前时间 */
thd->set_time();
/* other code ... */
}

```

想必有些同学已经清楚了，其实 Binlog 事件中的时间戳是从语句那里继承过来的，一条语句产生多个事件，那这些事件的时间戳都是一样的，而且都是和第一个事件一致的（这点可以自己验证一下）。

那上面关于 `exec_time=5` 的问题，该怎么解释呢？再来看一下代码，如下。

```

Log_event(
    thd_arg,
    (thd_arg->thread_specific_used ? LOG_EVENT_THREAD_SPECIFIC_F :
    0) |
    (suppress_use ? LOG_EVENT_SUPPRESS_USE_F : 0),
    using_trans ? Log_event::EVENT_TRANSACTIONAL_CACHE :
        Log_event::EVENT_STMT_CACHE,
    Log_event::EVENT_NORMAL_LOGGING,
    header(), footer()),
    data_buf(0)
{
    /* local vaiables */

    /* exec_time calculation has changed to use the same method that is used
    to fill out "thd_arg->start_time" */

    struct timeval end_time;
    ulonglong micro_end_time= my_micro_time();
    my_micro_time_to_timeval(micro_end_time, &end_time);

    /* 时间的计算，是用当前时间（执行完成的时间），减去thd_arg->start_time
    的值，这个值在上面已经见过，就是语句开始执行的时间，也就是说，exec_time
    指的就是语句从开始到结束所用的时间，即实际上的语句执行时间 */

    exec_time= end_time.tv_sec - thd_arg->start_time.tv_sec;
    /* other code ... */
}

```

那么现在就可以去解释事务 83631679 为什么执行了 5 秒钟，但所有事件的时间都是 10:11:30 了。就是因为这个事务是自动提交的，只有一条语句并且执行了 5 秒钟，就是这么简单，仅此而已。

因为是自动提交的，这个事务只有一条语句，`thd->set_time()` 也只会被设置一次，所以这个事务中的所有事件，都停留在了这个时间点上，所以就出现了上面的现象。

发散思维

可能有同学有疑惑了，即使一个事务只有一条语句，那也是有提交的，提交时间确实是在 5 秒之后做的，难道内部没有做这个问题的处理？可以做一个实验来看一下，还是一个事务一条语句，但此时不是自动提交，而是一个事务开始之后，等待了 5 秒钟，再去手动提交，然后再看 Binlog 内容，如下。

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into my1 values(13143306,'zhufeng');select sleep(5);commit;
Query OK, 1 row affected, 0 warning (0.00 sec)

+-----+
| sleep(5) |
+-----+
|          0 |
+-----+
1 row in set (5.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

为了没有时间误差，上面的三条语句是同时执行的，中间做了 5 秒的等待操作，现在看一下对应的 Binlog 内容。

```
#161216 12:53:25 server id 23932 end_log_pos 449 CRC32 0x2939a45b
GTID [commit=yes]
#161216 12:53:20 server id 23932 end_log_pos 522 CRC32 0x1df41360
Query thread_id=21 exec_time= error_code=0
#161216 12:53:20 server id 23932 end_log_pos 614 CRC32 0xf1515ed0
Write_rows: table id 78 flags: STMT_END_F
#161216 12:53:25 server id 23932 end_log_pos 645 CRC32 0x69c2d142
Xid = 29321
```

此时很明显地看到了，事务的 Xid 及 GTID 两个提交事件的时间，都比执行插入的时间晚 5 秒钟。这正是因为，执行的 Commit 语句与 INSERT 语句一样，都是一条语句，在执行前都会执行 `thd->set_time()`，从而影响了自自己生成的 Binlog 事件。

事务中的事件顺序

上面已经了解过，在一个事务中，会有事务开始的事件、事务提交的事件，也会有真正做事的事件，比如 `Write_rows` 等，它们之间的顺序，会与时间戳有一点关系。细心的同学可能已经发现，上一小节举的例子中，GTID 在最前面，它的时间是 12:53:25，而 `Write_rows` 在中间，但它的时间是 12:53:20，这之间有什么关系么？

其实，这在之前介绍 MySQL 5.7 多线程复制原理的时候已经讲过，在 MySQL 事务提交时，做的操作有如下三部分。

- 根据执行后的上下文环境，生成一个 GTID 事件。
- 组装事务产生的 GTID。
- 提交到各种存储引擎。

从上面所做的事情中可以看到，GTID 信息其实是在提交时生成的。这一点在 MySQL 5.7 中会更加明确，因为还需要我们已经熟知的 `last_committed` 及 `sequence_number` 来构造 GTID（具体请参考第 15 章）。这就可以解释，为什么 GTID 这个事件对应的时间和 Xid 是一样的了，就是因为它的时间是从 Commit 语句那里继承过来的。当然，Xid 也是同样的道理了，因为它们是从同一个语句产生的两个不同事件。

但关于顺序问题，是与 GTID 这个事件的作用有关的。在 MySQL Binlog 中，必须要提前知道 GTID 的具体信息，所以在 MySQL 提交并组装对应的 Binlog 时将其放到了最前面，从而导致了目前看到的关于时间问题的现象。

问题延伸

再回过头来看一下，最开始等待 5 秒的案例如下。



```
#161213 10:11:30 server id 11766 end_log_pos 263690501 CRC32 0x6e798470
  GTID [commit=yes]
#161213 10:11:30 server id 11766 end_log_pos 263690592 CRC32 0x2b6a6d34
  Query thread_id=4900813 exec_time=5 error_code=0
#161213 10:11:30 server id 11766 end_log_pos 263691291 CRC32 0xc0f9ed87
  Update_rows: table id 17891 flags: STMT_END_F
#161213 10:11:30 server id 11766 end_log_pos 263691322 CRC32 0xe40764c4
  Xid = 83631679
```

为何一个更新语句执行了 5 秒钟？当然，可能有以下两个原因。

- 这个语句对应的查询条件，是一个慢查询，扫描会花很长时间（5 秒钟），结果只更新了一行（因为只有一个 `Update_rows` 事件），从而导致了上面的现象。

- 本身执行不慢，但在等其他事务或锁的执行操作时，等了 5 秒钟。

这两种原因都是有可能的，也都可以解释得通。确定是哪种原因导致问题的方法很简单，那就是查看慢查询日志文件，找到 `thread_id` 为 4900813 的慢查询，或者对应的表的慢查询，并且一定要在 `server_id` 为 11766 的实例上面（这一点每个人都知道，但有时候会被忽略掉）。如果找到了，那就是第一种原因；如果找不到，那就是第二种原因了。

找啊找，结果在那个时间段内，都没有慢查询。

不管什么原因，执行了 5 秒钟，肯定是慢查询，怎么能找不到呢？这里对于 MySQL 的慢查询记录要多说一点，锁等待的时间在这里是不计算在内的。所以，如果是第 2 种原因，那么慢查询就必然是查不到的，并且 `exec_time=5` 对这一点也很有说服力，因为执行时间的计算是从开始时间到结束时间的差值，和慢查询的计算方法不同，所以这也说明了这 5 秒钟时间都是在等待的。

那么问题来了，它在等谁呢？竟然能等 5 秒钟？想必有另一个事务，拿到了它想要的锁，并且拿到锁的时间肯定超过 5 秒钟，那就需要继续找了。此时已经知道，大事务有如下 2 种。

- 单条自动提交的语句，本身执行时间长（`exec_time` 比较大）。
- 事务开始时间和结束时间的跨度长。

找 ROW 格式的 Binlog，咱有利利器啊！来看下面一段脚本，这段脚本也会出现在第 42 章中。

```
# vim summarize_binlogs.sh
#!/bin/bash
BINLOG_FILE="mysql-bin.000046"
START_TIME="2015-06-28 8:45:00"
STOP_TIME="2015-06-28 10:10:00"
mysqlbinlog --base64-output=decode-rows -vv --start-datetime="${START_TIME}"
--stop-datetime="${STOP_TIME}" ${BINLOG_FILE} | awk \
'BEGIN {xid="null";s_type="";stm="";endtm="";intsta=0;inttal=0;s_count=0;
count=0;insert_count=0;update_count=0;delete_count=0;flag=0;bf=0;period=0;} \
{
if (match($0, /^(BEGIN/)) {bg=1;} \
if (match($0, /#.*server id/)) {if(bg==1){statm=substr($1,2,6)" "$2;cmd=
sprintf("date -d \"%s\" +%s", statm);cmd|getline intsta;close(cmd);bg=0;bf=1;}
else if(bf==1){endtm=substr($1,2,6)" "$2;cmd=sprintf("date -d \"%s\" +%s",
endtm);cmd|getline inttal;close(cmd);}} \
if(match($0, /#.*Table_map:.mapped to number/)) {printf "Timestamp : " $1 " "
$2 " Table : " $(NF-4); flag=1} \
else if (match($0, /#.*Xid =./)) {xid=$(NF)} \
else if (match($0, /(### INSERT INTO .*.*))){count=count+1;
```

```

insert_count=insert_count+1;s_type="INSERT"; s_count=s_count+1;} \
else if (match($0, /(### UPDATE *.*.*/)) {count=count+1;
update_count=update_count+1;s_type="UPDATE"; s_count=s_count+1;} \
else if (match($0, /(### DELETE FROM *.*.*/)) {count=count+1;
delete_count=delete_count+1;s_type="DELETE"; s_count=s_count+1;} \
else if (match($0, /^(# at) /) && flag==1 && s_count>0) {print " Query Type : "
s_type " " s_count " row(s) affected" ;s_type=""; s_count=0; } \
else if (match($0, /^(COMMIT)/)) {period=inttal-intsta;if(inttal==0){period=0};
print "[Transaction total : " count " Insert(s) : " insert_count
" Update(s) : " update_count " Delete(s) : " \
delete_count " Xid : "xid" period : "period" ] \n+-----+
-----+-----+-----+"; \
count=0;insert_count=0;update_count=0; delete_count=0;s_type=""; s_count=0;
flag=0;bf=0;bg=0;} } '

```

这段脚本可以帮我们很轻松地找到跨度比较长的事务，通过结果中的 period 列来表示。分析了 period 列的数据之后，结果如下。

```

zhufeng@zhufengmac:~$ cat sum|grep Transaction|awk '{if($19>0)print}'
[Transaction total : 8 Insert(s) : 8 Update(s) : 0 Delete(s) : 0 Xid : 83631021
period : 1 ]
[Transaction total : 15 Insert(s) : 10 Update(s) : 4 Delete(s) : 1 Xid : 83631137
period : 1 ]
[Transaction total : 10 Insert(s) : 7 Update(s) : 3 Delete(s) : 0 Xid : 83631517
period : 1 ]
[Transaction total : 3 Insert(s) : 3 Update(s) : 0 Delete(s) : 0 Xid : 83631658
period : 1 ]
[Transaction total : 16 Insert(s) : 10 Update(s) : 6 Delete(s) : 0 Xid : 83631678
period : 51 ]
[Transaction total : 12 Insert(s) : 9 Update(s) : 3 Delete(s) : 0 Xid : 83631829
period : 1 ]
[Transaction total : 5 Insert(s) : 4 Update(s) : 1 Delete(s) : 0 Xid : 83632139
period : 1 ]
[Transaction total : 3 Insert(s) : 3 Update(s) : 0 Delete(s) : 0 Xid : 83632172
period : 1 ]
[Transaction total : 3 Insert(s) : 3 Update(s) : 0 Delete(s) : 0 Xid : 83632206
period : 1 ]

```

很快发现，Xid 为 83631678 的事务，竟然长达 51 秒。然后打开对应的 Binlog 文件找到这个事务，内容如下。



```
#161213 10:11:35 server id 11766 end_log_pos 263677208 CRC32 0xbfc41688
  GTID [commit=yes]
#161213 10:10:44 server id 11766 end_log_pos 263677291 CRC32 0x02537685
  Query   thread_id=4901481  exec_time=0 error_code=0
#161213 10:10:44 server id 11766 end_log_pos 263677435 CRC32 0x0e70aab6
  Write_rows: table id 17852 flags: STMT_END_F
#161213 10:10:44 server id 11766 end_log_pos 263677609 CRC32 0xb58d4c61
  Update_rows: table id 17853 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263685326 CRC32 0xc512e73c
  Write_rows: table id 566 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263685556 CRC32 0x805318e4
  Write_rows: table id 566 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263690422 CRC32 0x4de989c8
  Write_rows: table id 73 flags: STMT_END_F
#161213 10:11:35 server id 11766 end_log_pos 263690453 CRC32 0xbec3aaf5
  Xid = 83631678
```

发现什么了吗？这不是事务 83631678 么，是本章开始位置所展示出来的三个事务中的第一个？没错，正是它。

首先，关于 GTID 事件和 Xid 事件的时间问题，上面已经解释过了，这是提交语句的时间，所以都是 10:11:35，先忽略它。而中间真正做事的一段内容，是需要重点关注的。前面两个事件，分别是 Write_rows 和 Update_rows，它们的时间是 10:10:44，而后面三个 Write_rows 事件的时间是 10:11:35，中间长达 51 秒钟，这段时间做什么了？

再核对一下事务 83631679 中的 Update_rows 要修改的表的记录，与事务 83631678 中在 10:10:44 时间点发生的事件 Update_rows 所要修改的记录，是同一个表中的同一条记录。那肯定是要等待了。

数据库问题，都已经解释清楚了，现在唯一的问题，就是需要找到业务开发人员，问一句，那个事务在哪个表上，在那 51 秒钟的时间里做什么了？

DBA 还要继续处理其他问题，接力棒现在就交给开发同学了，明天听你回话。

show processlist 中的 Time

下面的问题，可能是在实际运维过程中遇到的容易造成疑惑的问题，先来看看我们所熟知的 show processlist 结果，这里请重点关注结果中的 Time 列信息，如下。



```
mysql> show processlist\G
***** 1. row *****
```

```

      Id: 1
      User: zhufeng.wang
      Host: localhost:51703
      db: NULL
      Command: Sleep
      Time: 5142
      State:
      Info: NULL
      Rows_sent: 0
      Rows_examined: 0
***** 2. row *****
      Id: 2
      User: zhufeng.wang
      Host: localhost:62898
      db: local
      Command: Sleep
      Time: 2077
      State:
      Info: NULL
      Rows_sent: 2
      Rows_examined: 2

```

上面两行信息中，Time 值分别是 5142 和 2077，它们是怎么来的呢？对于这个问题，各位同学应该都是比较清楚的，它代表的是当前语句在执行时的时间点，与执行 show processlist 命令时的时间差，从下面的 MySQL 代码中可以证明这一点。

```

/* 用来计算Show Processlist中的Time列的值，thd_info->start_time
   代表线程thd_info执行最后一个语句时的开始时间 */
if (thd_info->start_time)
    protocol->store_long ((longlong) (now - thd_info->start_time));
else
    protocol->store_null();

```

现在可以做一个实验，开两个会话，连接同一个 MySQL 服务器，一个会话（会话 1）用来做实验，另一个会话（会话 2）用来不断地执行 show processlist 命令以观察现象，实验结果很明确，只要会话 1 保持无操作，在会话 2 的结果中，会话 1 的连接对应的 Time 值一直在增长，而只要会话 1 执行任意一个命令，会话 2 的结果中，会话 1 的连接对应的 Time 值会被清零一次，然后再次继续增长，如此往复循环。在了解上述内容之后，这个现象现在应该很容易理解了，因为在执行任意命令时（调用函数 dispatch_command），都会执行 thd->set_time()，将时间清为当前时间，即时间差被清零。

但是有没有见过如下这样的现象？

```
mysql> show processlist\G
```

```
***** 1. row *****
      Id: 2
      User: zhufeng.wang
      Host: localhost:62898
      db: local
      Command: Query
      Time: 1203070
      State: User sleep
      Info: select sleep(100)
      Rows_sent: 0
      Rows_examined: 0
      ...其他省略...
```

可以看到,此时会话 1 的 Time 值高达 1203070,而对应的语句只是 `select sleep(100)`。是不是感到很奇怪,为什么只睡了 100 秒,而 Time 可以那么高?

当然,这里的语句是自己构造的,同时还发现,不管这样的语句执行多少次,Time 依然保持上涨,并不会清零,这是什么原因?

很明显,这样的现象会给 DBA 造成一些困惑,在解决问题时会造成干扰,所以有必要在这里解释清楚。下面重点看一下 `set_time` 这个函数的实现。

```
inline void set_time()
{
    /* 获取当前timestamp,存到start_untime变量中 */
    start_untime= utime_after_lock= my_micro_time();
    if (user_time.tv_sec || user_time.tv_usec)
    {
        /* 这里发现,当user_time不为0时,上面获取到的timestamp直接被忽略了,而是使用
           了user_time的值,也就是说,只要user_time的值不变,那么set_time的操作就不会
           改变当前连接的最新时间值,我们就需要研究清楚, user_time到底是什么! */
        start_time= user_time;
    }
    else
        /* 正常路径下,如果user_time为0,则更新当前连接最新时间*/
        my_micro_time_to_timeval(start_untime, &start_time);
}
```

上面代码中提到的参数 `user_time`,实际上对应的是 MySQL 会话参数 `timestamp`,只要显式地设置了这个参数, `user_time` 就不为 0,那么当前会话的起始时间就被固定在这一刻了。

现在明白了,只要在会话 1 中执行一条命令,比如 `set timestamp=1490264145;`,然后在会话 2 中观察,就会发现,Time 不仅非常大,而且在会话 1 中再执行任何语句时,会话 2 中的

Time 值都不会被清零了。

所以，如果在实际运维中遇到这样的问题，就可以找一下有没有连接执行过这样的语句，从而造成了这样的假象，因为这样的问题出现时，都会把这类语句误判为慢查询，而实际又找不到这样的查询。

这个问题是不是有种似曾相识的感觉？没错，在 Binlog 里经常会遇到这样的命令，这是 MySQL 为了保持主从执行环境的一致性而做的，但如果在主库上这样操作，经常是不仅不好玩，反而会造成一头雾水的感觉。

总结

这个问题，看似简单，实则涉及很多关于 Binlog 的细节问题。讲这些的主要目的就是让 DBA 同学了解时间戳在 Binlog 中的作用及产生方法，以便在出现一些这方面怪异的问题时，做到心中有数，胸有成竹。

22

InnoDB 中 Rowid 对 Binlog 的影响

背景

前段时间，有一个 MySQL 同行遇到一件事，苦思冥想好几天一直都没有解决。他用的是 Galera Cluster 方案，说是这套方案有问题，总是出现卡死的情况。

他描述了一下这个问题，大概是这样的情景。在某一个节点中，一条 SQL 语句，更新了数据，影响了大概 10 万行数据，然后就出现 Flow Control，整个集群卡死了，时间长达 3 个小时，迟迟不能恢复，在这种情况下，只好将发送流量控制的节点杀死才能解决问题。这样的问题出现过好几次，每次都这样解决，但不知道为什么会出现这样的问题。

这种解决办法是非常正确的，卡死之后，尽快找到一直发送流量控制的那个或那些节点，赶紧杀死，这样才能让集群恢复服务。什么？哪些？会出现这种现象的多个节点么？

没错，这个问题就是需要杀死两个节点（假如集群中是三个节点的话）才能解决问题，因为两个节点同时都在发送 Flow Control，太可怕了。

问题分析

这是为什么呢？“一条语句下去，其他节点都发生流量控制，那是不是这个表没有主键呢？”我直接问道。在线等结果，一会儿，那同学很惊奇地告诉我，“果然没有！”还好，在第 37 章中也会说到这是原因之一。

但是这能怪 Galera Cluster 么？如果是简单的主从复制，在一个表没有主键，Binlog Format 又是 ROW 的情况下，主从延迟还不会非常大，即使一直跟不上，也只是主从异步而已，对主库的影响不大，仅此而已。在这里还是想说，Galera Cluster 固然有其问题，但了解其原因并有效地避免，才是最好的。

当然这里顺带说一句，除了没有主键不应该之外，在 Galera Cluster 上面最好不要让一个事务更新那么多数据，可以适当控制在 1 万行以内，都是没有问题的，因为之前已经说过，Galera Cluster 的验证和提交是串行的，一个事务太大，会导致集群其他事务都等待这个事务完成，造成集群假死现象。

现在再回到这个问题上来，此时有另外一位同学问了这么一句：“但是，InnoDB 如果没有指定主键，就会创建一个 Rowid，那为什么在从库中更新 10 万行数据还那么慢呢？”我的回答是：“MySQL 的 Binlog 是 Server 层的东西，而 InnoDB 中的 Rowid 是存储引擎的东西，Server 层根本感知不到 Rowid 的存在，那么复制时从库如何知道哪一行对应哪一个 Rowid 呢。”这句话似乎是老生常谈了，但总感觉有点官方，让一个人理解一些东西，最好还是拿出一些实证来，实实在在地看到，才能更好地理解，那么就想办法证明一下吧。

首先，创建一个没有主键的表：



```
mysql> show create table t\G
***** 1. ROW *****
      Table: t
Create Table: CREATE TABLE `t` (
  `sno` int(11) DEFAULT NULL,
  `t` timestamp NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

然后，插入 5 条数据：



```
mysql> insert into t (sno, t) values(1,now()),(1,now()),(1,now()),(1,now()),
(1,now());
Query OK, 5 rows affected (0.00 sec)
Records: 5 Duplicates: 0 Warnings: 0
```

验证插入的数据：



```
mysql> select * from t;
+-----+-----+
| sno | t      |
+-----+-----+
| 1   | 2016-12-02 12:32:58 |
```

```
| 1 | 2016-12-02 12:32:58 |
| 1 | 2016-12-02 12:32:58 |
| 1 | 2016-12-02 12:32:58 |
| 1 | 2016-12-02 12:32:58 |
+-----+-----+
5 rows in set (0.00 sec)
```

此时，可以看一下 Binlog 中的内容，看一下里面是不是有 Rowid 信息，如图 22.1 所示。

```
BINLOG '
2/dAWBMJAAAAAgAAABTaTh8AAFoBAAAAAAEABHRLc3QAAXQAAgMRAQADKR8BVw==
2/dAWB4JAAAAUAAAAAGTaTh8AAFoBAAAAAAEAAgAC//wBAAAAWED32/wBAAAAWED32/wBAAAAWED3
2/wBAAAAWED32/wBAAAAWED321TChB0=
'/*!*/;
### INSERT INTO `test`.`t`
### SET
### @1=1 /* INT meta=0 nullable=1 is_null=0 */
### @2=1480652763 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
### INSERT INTO `test`.`t`
### SET
### @1=1 /* INT meta=0 nullable=1 is_null=0 */
### @2=1480652763 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
### INSERT INTO `test`.`t`
### SET
### @1=1 /* INT meta=0 nullable=1 is_null=0 */
### @2=1480652763 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
### INSERT INTO `test`.`t`
### SET
### @1=1 /* INT meta=0 nullable=1 is_null=0 */
### @2=1480652763 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
### INSERT INTO `test`.`t`
### SET
### @1=1 /* INT meta=0 nullable=1 is_null=0 */
### @2=1480652763 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
# at 525261412
#161202 12:26:03 server id 9 end_log_pos 525261443 CRC32 0x279010a0 Xid = 204301921
COMMIT/*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
DELIMITER ;
# End of log file
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

图 22.1

从这些内容中，并没看出来有 Rowid 信息，但可能有人还是怀疑，那前面的二进制中，有没有解析并展示出来的内容呢？

那再想其他办法证明。不过可以先想一下，如果 Binlog 里面有 Rowid，那么主库所更新的数据，从库更新的必然是同样的数据。

目前的情况是，上面的 5 条数据中，数据是完全一样的。在没有主键的情况下，构造一个场景，如果可以看到主库和从库更新的是同样位置的记录（不能是第一行，原因可想而知），那就说明 Binlog 里面有 Rowid 了，否则就是没有。写一条语句来做这件事，如下。

```
mysql> update t set t = now() order by rand() limit 1;
```

因为针对一个表中数据全部是一样的情况，没办法写出一条语句来更新指定某一行的数据，所以只好使用 rand 这个神器了，更新之后，主库表中的数据如下。

```
mysql> select * from t;
+-----+-----+
| sno | t                |
+-----+-----+
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:33:03 |
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:32:58 |
+-----+-----+
5 rows in set (0.00 sec)
```

可以发现，这次如我们所愿，更新的正是第三条记录，其他的还是原来的值。此时从库中这个表的数据如下。

```
mysql> select * from t;
+-----+-----+
| sno | t                |
+-----+-----+
| 1   | 2016-12-02 12:33:03 |
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:32:58 |
| 1   | 2016-12-02 12:32:58 |
+-----+-----+
5 rows in set (0.00 sec)
```

这正是预计要看到的内容，不过需要先强调一下，因为这个表没有任何索引，查询全表必然是按照物理上从前到后的顺序进行扫描，所以这个顺序也正是表中的物理顺序，同时也是逻辑顺序。

此时，主库和从库的数据在更新之后，顺序是不同的。这是因为主库更新的数据行在传到从库之后，发现从库的第一行可以被匹配，便随即更新了第一行的数据，所以造成了这样的问题。

假设主库中的 5 条数据对应的 Rowid 分别是 1、2、3、4、5。Rowid 为 3 的记录时间为“2016-12-02 12:33:03”，而在从库上，Rowid 为 1 的记录时间为“2016-12-02 12:33:03”，从物理意义上说，这就造成了数据的不一致问题。

这就更有力地证明了 Binlog 其实是关心不到 InnoDB 存储引擎的 Rowid 值的，爱莫能助！

当然，如果真的能关心到了，那么这个问题自然也就解决了。但有些同学可能有意见了，这有关系么？数据从逻辑上又不会出现什么错误，也没什么影响，只是看上去顺序不同，同一个语句使用不同索引的时候查出来的数据顺序还有可能不同呢，没什么大不了的。

是的，的确如此！但这一章的内容，其意义并不在这里，而是通过这样的方式，让初学者，或者已经入门但没有想过这些问题的同学，知道一些关于 MySQL 的细节问题而已，也算是思维发散吧。

再看一下更新时的 Binlog 内容，如图 22.2 所示。

```
f/1AWBMJAAAAIgAAAGbeTh8AAFoBAAAAAAEABHRIc3QAAXQAAGMRAQADUowN1w==
f/1AWB8JAAAANGAAAjzeTh8AAFoBAAAAAAEAAgAC///8AQAAAFhA+Xr8AQAAAFhA+X8UncrZ
'/*!*/;
### UPDATE `test`.`t`
### WHERE
###   @1=1 /* INT meta=0 nullable=1 is_null=0 */
###   @2=1480653178 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
### SET
###   @1=1 /* INT meta=0 nullable=1 is_null=0 */
###   @2=1480653183 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
#at 525262492
#161202 12:33:03 server id 9  end_log_pos 525262523 CRC32 0xa4fb8bf5  Xid = 204301938
COMMIT/*!*/;
```

图 22.2

那不是写着“UPDATE test.t WHERE @1=1 @2=1480653178 SET @1=1 @2=1480653183”么？在从库中怎么只更新了一行？可以先问一下自己是不是知道原因。

简单说一下这个问题。对于 Row 模式而言，一个 DML 事件，针对的是一行数据，而不是一个语句，通过 mysqlbinlog 工具看到的只是一个可以让人看得更清楚的表象而已，而不是一个等价的语句，所以在实际执行时，只要更新了一条，就结束了，不会再向后扫描了。

此时可以想象一下，如果上面的语句更新的行数是 2，而不是 1，那会出现怎样的情景，直接通过主从来对比，同样的数据在改之前还是 5 条完全一样的数据，改之后主库数据如下。

```
mysql> select * from t;
+-----+-----+
| sno  | t      |
+-----+-----+
| 1    | 2016-12-02 12:55:40 |
```

```
| 1 | 2016-12-02 12:55:28 |
| 1 | 2016-12-02 12:55:28 |
| 1 | 2016-12-02 12:55:28 |
| 1 | 2016-12-02 12:55:40 |
+-----+-----+
5 rows in set (0.00 sec)
```

从库数据如下。

```
mysql> select * from t;
+-----+-----+
| sno | t |
+-----+-----+
| 1 | 2016-12-02 12:55:40 |
| 1 | 2016-12-02 12:55:40 |
| 1 | 2016-12-02 12:55:28 |
| 1 | 2016-12-02 12:55:28 |
| 1 | 2016-12-02 12:55:28 |
+-----+-----+
5 rows in set (0.00 sec)
```

确实是这样的，因为在主库中是随机位置更新的，但是在从库中是一条一条地更新，从前到后扫描，定位当然是最开始位置，而不会最先更新到后面的数据。而更严重的是，如果一个表中的数据很多，由于是全表扫描，针对每一条 ROW 复制的 APPLY，会导致越是 APPLY 到后面，扫描的数据就越多，性能也就越差，因此表越大这种问题就越明显。

总结

通过这个看上去没有实际意义的例子，可以明白以下 4 个问题。

- Binlog 只是 Server 层的东西，需要兼顾 MyISAM 等其他存储引擎，而 Rowid 是 InnoDB 的东西，所以它是不能感知的。
- Rowid 是 InnoDB 自己定义的一个列，只有在表中没有定义主键的时候，系统才会给这个表加上这一列，但这一列只是为了存储，构造成一个聚簇索引，但不会暴露给逻辑层，并且上层也用不到它，所以直接忽略它即可。
- 在 MySQL 数据库的使用中，一定要定义主键，如果没有主键，并且是 Row 模式的复制，就必然会造成这样的问题，而不像其他数据库一样，如果没有主键，还可以使用 Rowid 来操作表。
- 在 Galera Cluster 中，更要定义主键。如果没有定义，势必会造成故障，这不是 Galera Cluster 的问题，只是它会把这个问题放大而已。

MySQL 备份:

Percona XtraBackup 的原理与实践

备份背景及类型

对于 DBA 来说,数据库的备份与恢复是日常运维中最常见的工作之一。在数据库运维的过程中,经常会遇到数据库服务器宕机、磁盘损坏等情况。在这种情况下,要保证数据不丢失,或者最小程度的丢失,备份起着至关重要的作用。备份工具的使用与备份系统的完善,以及备份是否正常工作都是每一位 DBA 时刻需要关注的。

在数据库的备份中,可以选择不同的方式进行备份。

根据备份方法,备份可以划分为如下三种。

- 热备份:热备份是指在数据库运行的过程中进行备份,对生产环境中的数据库运行没有任何影响。常见的热备方案是利用 `mysqldump`、`xtrabackup` 等工具进行备份。
- 冷备份:冷备是指在数据库关闭的情况下进行备份,这种备份非常简单,只需要关闭数据库,复制相关的物理文件即可。目前,线上数据库一般很少能够接受关闭数据库,所以该备份方式很少使用。
- 温备份:温备份也是在数据库运行的过程中进行备份,但是备份会对数据库操作有所影响。该备份利用锁表的原理备份数据库,由于影响了数据库的操作,故该备份方式也很少使用。

根据备份文件的种类，备份可以划分为如下 2 种。

- 物理备份：物理备份是指复制数据库的物理文件。物理备份即可以在数据库运行的情况下进行备份（常见备份工具：MySQL Enterprise Backup（商业）、XtraBackup 等），也可以在数据库关闭的情况下进行备份。该备份方式不仅备份速度快，而且恢复速度也快，但是由于无法查看备份后的内容，所以只能等到恢复之后，才能检验备份出来的数据是否是正确的。
- 逻辑备份：逻辑备份是指备份文件的内容是可读的，该文本一般都是由一条条 SQL 语句或者表的实际数据组成。常见的逻辑备份方式有 `mysqldump`、`SELECT * INTO OUTFILE` 等方法。这类备份方法的好处是可以观察备份后的文件内容，缺点是恢复时间往往都会很长。

根据备份内容，备份可以划分为如下三种。

- 全量备份：全量备份是指对数据库进行一次完整的备份，备份所有的数据。这是一般常见的备份方式，可以使用该备份快速恢复数据库，或者搭建从库。恢复速度也是最快的，但是每次备份会消耗较多的磁盘空间，并且备份时间较长。
- 增量备份：增量备份是指基于上次完整备份或增量备份，对数据库新增的修改进行备份。这种备份方式有利于减少备份时使用的磁盘空间，加快备份速度。但是恢复的时候速度较慢，并且操作相对复杂。
- 日志备份：日志备份是指对 MySQL 数据库二进制日志的备份。该备份方式一般与上面的全量备份或增量备份结合使用，可以使数据库恢复到任意位置。

上面的多种备份方式，每个都有自己的优缺点。但是在生产环境中，备份一般需要满足以下三个需求。

- 备份时，数据库需要提供服务，即需要热备份。
- 生产环境中，需要随时恢复到某个时间点。便于线上出现问题后的数据回滚。
- 可以用来快速搭建新的从库，建立复制关系，实现数据的高可用、负载均衡的目标。

考虑到上述需求，一般都会选择物理备份为主，逻辑备份为辅，加上日志备份，来满足线上使用数据库的需求。物理备份的第三方工具有很多，本章主要介绍 Percona XtraBackup 2.4.4 工具的使用、特性、备份原理（老版本备份过程有所不同）以及使用中的注意事项。

认识 Percona XtraBackup

在众多针对 MySQL 备份的工具里面，XtraBackup 无疑是最流行的一种解决方案。到本书写作时间为止，它已经升级到了 2.4.4 版本，而最早开始使用 XtraBackup，要追溯到 0.9 版本。

经历了多年的发展和无数生产实践的验证, XtraBackup 已经被 MySQL DBA 们广泛接受, 并在生产实践中大规模使用。

XtraBackup 是由 Percona 公司开发的一款开源的 MySQL 热备工具, 在备份的过程中不会被锁在数据库中。该工具能够备份 InnoDB、XtraDB、MyISAM 表, 甚至可以备份 Percona XtraDB Cluster。Percona XtraBackup 支持 Percona Server 的所有版本, 支持 MySQL, 支持 MariaDB 的备份。并且还支持流方式、压缩、加密和增量备份等。

Percona XtraBackup 的优点如下。

- 无须停止数据库进行 InnoDB 热备。
- 增量备份 MySQL。
- 流压缩传输到其他服务器。
- 在线移动表。
- 能够比较容易地创建主从同步。
- 备份 MySQL 时不会增大服务器负载。

在这套工具集里最重要的程序有两个: innobackupex 与 xtrabackup, 前者是 perl 脚本, 后者是 C/C++ 编译的二进制程序。xtrabackup 是用来备份 InnoDB 的, 并不能备份非 InnoDB 表, 它在内部实现了对 InnoDB 的热备份。innobackupex 脚本是对 xtrabackup 的封装, 通过调用 xtrabackup 命令来备份 InnoDB 表, 同时也通过 mysqldump 等命令来备份非 InnoDB 表, 并且会与 MySQL 数据库发送命令交互, 例如获取 Binlog 位点、添加锁操作。

由于 MySQL 中的系统表 (例如: mysql 库) 基于 MyISAM 存储引擎 (MySQL 5.7 版本之前), 所以备份这部分数据一般都使用 innobackupex 脚本来完成。

由于 innobackupex 会调用 xtrabackup, 并把它获得的一些执行参数传递给 xtrabackup, 所以在介绍二者的时候, 可能会感觉有重复的部分。下面会从上到下逐步说明二者的用法。

在本章中, 用 XtraBackup 来表示 Percona 的这套备份工具, 而用 xtrabackup 来表述最核心的 innodb 引擎表的备份程序, 同样地, innobackupex 是表述针对所有引擎表的备份程序。

XtraBackup 的工作流程

首先, 来描述一下 XtraBackup 的工作流程。在工作过程中, innobackupex 会和 xtrabackup 协同工作, 共同完成备份任务。

- innobackupex 启动后, 首先通过 start_ibbackup() 函数中的 fork 方式创建 xtrabackup 进程并且启动。然后等待 xtrabackup 完成 InnoDB 相关文件的备份。
- innobackupex 通过 wait_for_ibbackup_suspend(\$suspend_file) 检测 xtrabackup_suspended_2

文件是否存在, 如果检测到该文件, innobackupex 进程就会被唤醒。

- xtrabackup 在备份 InnoDB 相关文件时, 会开启如下两种线程。
 - ibd 复制线程, 负责复制 ibd 文件。
 - redo log 复制线程, 负责复制 redo log 信息。xtrabackup 首先启动 redo log 复制线程, 从最近的 checkpoint 点开始顺序复制 redo log; 然后启动 ibd 复制线程, 在 xtrabackup 复制 ibd 的过程中, redo log 复制线程一直工作, 并且 innobackupex 进程处于等待状态, 等待被 xtrabackup 进程唤醒。
- xtrabackup 复制 ibd 完成后, 通知 innobackupex 进程 (通过创建 xtrabackup_suspended_2 文件方式), 同时自己进入等待状态, 但是 redo log 复制线程依然在工作。
- innobackupex 收到通知, 会执行备份锁 (LOCK TABLES FOR BACKUP), 取到一致性的位点, 然后开始复制非 InnoDB 文件。
- 当非 InnoDB 文件复制完成后, innobackupex 开始执行 LOCK BINLOG FOR BACKUP, 开始通过 get_mysql_master_status(\$con) 函数来获取 Binlog 位置。
- 然后创建 xtrabackup_binlog_info 文件, 通过 write_to_backup_file(\$binlog_info, join("\t", @binlog_info_content) . "\n") 函数将 Binlog 的位点信息写入文件中。
- 接着发起一个通知给 xtrabackup 进程 (通过删除 xtrabackup_suspended_2 的方式), 同时自己进入等待状态。
- xtrabackup 收到通知后, 就会停止 redo log 复制线程, 告知 innobackupex, redo log 复制完成, 然后通知 innobackupex (通过创建 xtrabackup_log_copied 文件的方式)。
- innobackupex 收到通知后, 就开始释放锁资源 (UNLOCK BINLOG 和 UNLOCK TABLES)。
- 随后, innobackupex 和 xtrabackup 进行后期工作, 比如资源的释放、备份元数据信息、打印备份目录、备份 Binlog 的位置信息, 以及写入 xtrabackup_info 文件信息等。
- 最后, innobackupex 等待 xtrabackup 进程结束后退出。

总结如下三点。

- 在备份的过程中, 需要复制 redo log, 是由于备份 ibd 文件的过程中该文件可能被修改, 这样备份出来的文件就可能是有脏数据的。那么在恢复时, 需要通过 redo log 进行数据恢复, 即应用已经提交的事务, 回滚那些未提交的事务。
- 在备份中严重依赖于 xtrabackup_suspended_2 和 xtrabackup_log_copied 文件, 如果在备份过程中, 人为地删除了该文件或创建了任何一个文件, 都会导致备份出现错误。因为 innobackupex 与 xtrabackup 进程之间的交互是通过这两个文件进行的。
- 如果是增量复制, 还存在 xtrabackup_suspended_1 文件, 作用是一样的。

XtraBackup 的备份原理

XtraBackup 是基于 InnoDB 自身的崩溃恢复机制完成备份的，它首先复制所有的 InnoDB 表数据文本副本，这样导致了内部数据的不一致。但是，它后来会执行崩溃恢复功能，使数据达到最终的一致，成为可用的数据库文件。而 innobackupex 会在外围打扫战场，解决除了 InnoDB 表和 redo log 之外的其他资源。

xtrabackup 之所以可以通过复制 InnoDB 表的 ibd 文件工作，是由于 InnoDB 内部维护了一个重做日志 (redo log)。这包含了每次 InnoDB 更改数据的记录。当 InnoDB 启动时，它会检查数据文件并重做日志，并执行以下两步操作：应用已经提交的事务（前滚），回滚那些未提交的事务（回滚）。

xtrabackup 在备份的时候，会记录 LSN (log sequence number) 位置，然后开始复制数据文件。如果数据库文件被修改了，数据库就会处于不一致的状态。与此同时，xtrabackup 开启一个后台进程，监控 redo log 的变化，并且复制变化的部分。xtrabackup 需要不断地循环该过程，因为 redo log 是循环使用的，可以重用。xtrabackup 需要记录每一次对数据文件操作的 redo log。

这样，从备份开始一直到备份结束，InnoDB 所有的 redo log 都能够备份，如果没有 redo log，那么由于数据是不一致的，后面就无法恢复了。

在外围，innobackupex 使用备份锁 (Backup Locks，一种相对 FLUSH TABLES WITH READ LOCK 来说更轻量级的锁) 工作。从 Percona Server 5.6+ 开始支持该特性，使用 LOCK TABLES FOR BACKUP 不会刷新表，因此 LOCK TABLES FOR BACKUP 仅等待冲突的语句 (DDL 和非事务表的更新) 完成。在此之前，innobackupex 使用 FLUSH TABLES WITH READ LOCK 自动复制非 InnoDB 数据。如果备份服务器支持备份锁，innobackupex 首先调用 xtrabackup 复制 InnoDB 表和 XtraDB 表数据，然后运行 LOCK TABLES FOR BACKUP，最后开始复制 MyISAM 表、.frm 文件和非事务表数据。此操作完成后，innobackupex 会继续备份 .frm、.MRG、.MYD、.MYI、.TRG、.TRN、.ARM、.ARZ、.CSM、.CSV、.par 和 .opt 文件。

在上述操作后，innobackupex 会执行 LOCK BINLOG FOR BACKUP 操作，锁住任何可能改变 Binlog 的信息，或者 Exec_Master_Log_Pos 和 Exec_Gtid_Set，获取 SHOW MASTER 或 SHOW SLAVE 信息。innobackupex 将完成对 redo log 的复制和 binary log 位置信息的获取。完成这步操作后，innobackupex 将会释放锁。备份 Binlog 的信息，是为了在备份结束时，获取数据库的位置。在搭建从库时，可以使用该数据库位置直接用于 CHANGE MASTER 语句中。

最后，二进制日志的位置信息将会输出到 STDERR 中，并且 innobackupex 和 xtrabackup 会退出。

在 prepare 阶段，XtraBackup 使用复制的 redo log 信息对复制的数据文件进行崩溃恢复，在此阶段后，备份的 InnoDB 表将最终相互一致。

在恢复的过程中, innobackupex 首先会读取 my.cnf 配置文件的变量, 包括 `datadir`、`innodb_data_home_dir`、`innodb_data_file_path`、`innodb_log_group_home_dir`, 并且会检查这些目录是否存在。然后开始复制 MyISAM 表、索引文件 (`.frm`、`.MRG`、`.MYD`、`.MYI`、`.TRG`、`.TRN`、`.ARM`、`.ARZ`、`.CSM`、`.CSV`、`.par` 和 `.opt` 文件) 等。再接着复制 InnoDB 表、索引文件, 最后复制日志文件。



注意: 这些文件在复制的时候, 将会保留文件属性, 所以在启动数据库服务器之前, 需要修改文件的所有权。

在恢复的过程中, 也可以使用 `--move-back` 选项, 该选项与 `--copy-back` 的唯一区别就是, 该选项是将备份文件移动到目标位置, 而不是复制, 即会删除备份文件, 所以使用时一定要小心。一般在磁盘空间不足以同时保存数据文件和备份文件时, 才使用该选项。

在备份完成后, 备份目录中会产生以下八个主要的文件。

- `backup-my.cnf`: 此文件包含了备份所需的 my.cnf 中的选项。例如: `innodb_data_file_path`、`innodb_log_file_in_group`、`innodb_log_file_size` 等。在恢复的过程中, innobackupex 会依赖于该文件的选项。
- `xtrabackup_checkpoints`: 该文件记录备份的类型 (`full-backup`ed、`incremental`)、备份状态及备份的 LSN 信息, 增量备份依赖于该信息。
- `xtrabackup_binlog_info`: 该文件记录数据库备份时的二进制文件和位置信息。这是备份后数据库的位置, 作用非常大, 当利用备份来搭建从库时, 该二进制信息可以直接用来在 `CHANGE MASTER` 中使用。并且, 如果通过它可以知道备份时数据库的位置, 那么可以与 Binlog 结合使用, 恢复到任意位置。
- `xtrabackup_binlog_pos_innodb`: 该文件记录了 InnoDB 表或 XtraDB 表的二进制文件和位置信息。当执行 `--apply-log` 时, 会创建该文件。
- `xtrabackup_binary`: 该文件记录备份进程使用的 xtrabackup 二进制文件。
- `xtrabackup_logfile`: 该文件记录 `--apply-log` 操作时所需的数据 (`ib_logfile0`)。如果 `--apply-log` 操作时间很长, 该文件就会很大。
- `xtrabackup_slave_info`: 如果备份时指定 `--slave-info` 选项, 就会产生该文件, 该文件记录了在建立主从关系时 `CHANGE MASTER` 语句所需的信息。
- `xtrabackup_galera_info`: 如果备份时, 执行 `--galera-info` 选项, 就会产生该文件, 该文件记录了 `wsrep_local_state_uuid` 和 `wsrep_last_committed` 状态的值。主要用于备份 Galera 集群和 Percona XtraDB Cluster。

XtraBackup 需要的权限

XtraBackup 在备份时, 需要 MySQL 的一些权限, 如果权限不对, 则不能正常备份, 所以这里单独拿出来重点说一下, 如下。

- RELOAD、LOCK TABLES: 在备份时, 需要执行 FLUSH TABLES WITH READ LOCK 和 FLUSH ENGINE LOGS, 然后开始复制数据。并且在使用 Backup Locks 时, 需要执行 LOCK TABLES FOR BACKUP 和 LOCK BINLOG FOR BACKUP。除非备份时, 执行 --no-lock 选项, 这样便不需要这两个权限了。
- REPLICATION CLIENT: 在备份时需要获得二进制日志的位置信息, 需要该权限。
- CREATE TABLESPACE: 使用 XtraBackup 进行恢复独立表空间的时候, 需要使用导入表空间, 这时候需要该权限。
- PROCESS: XtraBackup 在备份的时候, 利用该权限可以查看所有正在服务器端运行的线程情况。
- SUPER: 用在复制环境中开启和关闭 SLAVE 线程。
- CREATE: CREATE 权限用于创建 PERCONA_SCHEMA.xtrabackup_history 数据库和表。
- INSERT: INSERT 权限用于将写入历史记录到 PERCONA_SCHEMA.xtrabackup_history 数据库和表。
- SELECT: 当 innobackupex 使用 --incremental-history-name 或 --incremental-history-uuid 选项时, 可以使用 SELECT 权限查询 PERCONA_SCHEMA.xtrabackup_history 表的 innodb_to_lsn 的值来满足该特性。

举个授权的例子, 如下。

```
mysql>CREATE USER 'backup'@'localhost' IDENTIFIED BY 'backup_password';
mysql>GRANT RELOAD, LOCK TABLES,REPLICATION CLIENT ON *.* TO 'backup'@'localhost';
```

innobackupex 常用的备份选项说明

- --apply-log: 通过应用 BACKUP_DIR 目录中的 xtrabackup_logfile 的事务日志, 在 BACKUP_DIR 目录中准备备份, 并且创建新的事务日志。innobackupex --apply-log 默认使用 BACKUP_DIR 目录 backup-my.cnf 中的 InnoDB 配置, 如果指定了 --defaults-file 选项, 那就使用它指定的配置文件。在此上下文中的 InnoDB 配置是指影响数据格式的服务器变量, 如: innodb_page_size、innodb_log_block_size 等。innobackupex --apply-log 总是忽略位置相关的变量, 如 innodb_log_group_home_dir、innodb_data_file_path。所以, 在备份准备节点使用适用于数据的备份目录, 不会影响到外部目录。

- `--compact`: 该选项创建一个省略所有辅助索引页的压缩备份。
- `--compress`: 该选项表示 xtrabackup 备份时压缩 InnoDB 数据文件, 直接传递给 xtrabackup 备份子进程。
- `--compress-threads=#`: 该选项指定并行压缩的线程数, 直接传递给 xtrabackup 备份子进程。
- `--copy-back`: 将准备好的备份文件从备份目录复制到原始位置, 其原始位置目录必须为空, 否则报错 (除非指定 `--force-non-empty-directories` 选项)。
- `--database=LIST`: 该选项指定 innobackupex 备份的数据库列表, 接受需要备份的数据库列表文件的字符串参数或路径, 例如 “db1[table1],db2[table2]”。如果该选项未指定, 默认备份包含 MyISAM 和 InnoDB 表的所有数据库。如果列表过长, 可以通过文件指定, 可参考 `--tables-file` 选项。
- `--decompress`: 对于上述 `--compress` 选项压缩备份文件, `--decompress` 用于解压上述压缩后的备份文件, 可以通过 `--parallel` 选项并行解压。
- `--defaults-file=[my.cnf]`: 该选项接受一个字符串参数, 指定从该配置文件中读取 MySQL 的配置选项。该选项必须作为命令行的第一个选项。
- `--host=HOST`: 指定数据库的 IP 地址。
- `--incremental`: 该选项表示 xtrabackup 创建增量备份, 而不是全量备份。
- `--incremental-basedir=DIRECTORY`: 该选项接受字符串参数, 表示包含作为增量备份基本数据集的完整备份的目录。与 `--incremental` 选项一起使用。
- `--incremental-dir=DIRECTORY`: 该选项接受字符串参数, 指定增量备份与完整备份的目录, 以进行新的完整备份, 与 `--incremental` 选项一起使用。
- `--incremental-lsn=LSN`: 该选项接受用于增量备份的日志序列号 (LSN) 的字符串参数。与 `--incremental` 选项一起使用, 可用于替代选项 `--incremental-basedir`。
- `--kill-long-queries-timeout=SECONDS`: 该选项指定 innobackupex 在启动 FLUSH TABLES WITH READ LOCK 和杀死阻止它的查询之间的等待秒数。默认为 0 秒, 意味着 innobackupex 不会尝试杀死其他任何查询。为了使用该选项, innobackupex 用户必须有 PROCESS 和 SUPER 的权限。在 Percona Server 5.6+ 中, xtrabackup 将自动使用轻量级的备份锁作为 FLUSH TABLES WITH READ LOCK 的替代, 来复制非 InnoDB 数据, 以避免阻塞修改 InnoDB 表的 DML 语句。
- `--kill-long-query-type=all|select`: 为了取消阻塞全局锁, 该选项指定哪种类型的语句被杀。
- `--log-copy-interval=#`: 该选项指定日志复制线程执行检查之间的时间间隔, 以毫秒为单位。
- `--move-back`: 将之前备份的所有文件从备份目录移动到原始目录。此选项恢复后将删除备份文件, 请慎用。

- `--no-lock`: 该选项使用 `FLUSH TABLES WITH READ LOCK` 禁用表锁定。只有在所有表都是 InnoDB 的情况才使用它, 并且不关心备份的二进制位置。如果有任何正在执行的 DDL 语句或非 InnoDB 表上发生任何更新, 则不应该使用此选项, 否则可能导致不一致的备份。在 Percona Server 5.6+ xtrabackup 上, 将自动使用轻量级的备份锁作为 `FLUSH TABLES WITH READ LOCK` 的替代来复制非 InnoDB 数据, 以避免阻止修改 InnoDB 表的 DML 查询。
- `--no-timestamp`: 该选项阻止在备份根目录内创建时间戳的子目录, 指定时, 备份将在备份根目录中完成。
- `--parallel=NUMBER-OF-THREADS`: 该选项接受一个整型参数, 该参数指定 xtrabackup 备份子进程在备份文件时使用的线程数。



注意: 该选项是文件级别的, 即如果有多个 .ibd 文件, 它们会被同时复制。如果表存放在单个表空间文件中, 该选项则不起作用。该选项也允许同时解密或加密多个文件。

- `--password`: 连接数据库的密码。
- `--port`: 连接数据库的端口号。
- `--redo-only`: 在 prepare 阶段, 完全备份合并除最后一个增量之外的所有增量时, 使用该选项。迫使 xtrabackup 跳过“回滚”阶段, 只做一次“重做”。
- `--safe-slave-backup`: 与 `slave-info` 结合使用, 发起备份时, 会暂停 SLAVE 的 SQL Thread, 确保备份时没有临时表打开, 保证数据的一致性。备份结束后, SQL 线程会自动启动。如果 `Slave_open_temp_tables` 在 `--safe-slave-backup-timeout` 秒后未变成零, 则备份将失败。
- `--safe-slave-backup-timeout=SECONDS`: 该选项指定 `--safe-slave-backup` 应等待多长时间, 以使 `Slave_open_temp_tables` 变为零。默认时间为 300 秒。
- `--slave-info`: 该选项用于在从库上备份。它打印二进制日志的位置和主库的名称, 还将此信息以“CHANGE MASTER”命令的形式写入到 `xtrabackup_slave_info` 文件中, 可以通过此备份启动从实例, 并发出“CHANGE MASTER”命令来设置此主库的新从库。
- `--socket`: 该选项接受一个字符串参数, 可以指定使用 UNIX 套接字的方式连接到本地数据库。
- `--stream=STREAMNAME`: 该选项接受指定流备份执行格式的字符串参数, 备份将以指定的格式完成到 STDOUT。目前, 支持的格式有 `tar` 和 `xbstream`。`xbstream` 在 percona xtrabackup 的版本中可用。如果在该选项后指定路径, 则它被解释为 `tmpdir` 的值。
- `--tmpdir=DIRECTORY`: 该选项接受一个指定临时文件位置的字符串参数。它可以与 `--stream` 选项一起使用。使用该选项, 事务日志首先会存储到临时文件中, 然后流式传输

或复制到远程主机。如果未指定该选项, 则默认值为使用 my.cnf 配置读取的 tmpdir 值。


- --use-memory=#: 该选项指定 xtrabackup 在 prepare 阶段进行崩溃恢复的内存使用量, 以字节为单位, 支持的单位如 1MB、1M、1GB、1G。它仅与 --apply-log 选项一起使用。
- --user=USER: 备份时连接数据库的用户名。

XtraBackup 备份实践


全量备份

备份阶段

在全量备份时, 只需要指定连接数据库的参数和备份目录即可。

```
 innobackupex --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS  
/path/BACKUP_DIR/
```

在备份完成后, innobackupex 会输出以下信息。


```
 ***  
161205 09:32:16 innobackupex: Starting ibbackup with command: xtrabackup ***  
***  
innobackupex: Backup created in directory '/home/q/BACKUP_DIR/2016-12-05_09-32-16'  
innobackupex: MySQL binlog position: filename 'mysql-bin.000003', position 330  
161205 09:32:28 innobackupex: Connection to database server closed  
161205 09:32:28 innobackupex: completed OK!
```

通过输出信息可以看出, 其实 innobackupex 在执行时, 调用了 xtrabackup 脚本。并且备份文件会存储在一个以备份时间命名的子目录下。

Prepare 阶段

在创建备份后, 备份数据其实处于不可用状态。因为在 redo log 中可能存在未提交的事务和已经提交的事务, 需要通过准备阶段来使备份数据达到一致的状态。通过此阶段, 备份数据就可以用来恢复了。

在准备阶段, 需要指定选项 --apply-log 和备份路径。

```
 innobackupex --apply-log /path/BACKUP_DIR/2016-12-05_09-32-16
```

输出信息如下。

```
InnoDB: Shutdown completed; log sequence number 1626134
161205 09:44:33 innobackupex: completed OK!
```

如果状态为“completed OK”，则表明 innobackupex 执行完了所有所需的操作，数据达到一致状态。

恢复阶段

在 Prepare 阶段过后，如果需要用备份数据来恢复数据库，则只需要指定--copy-back 和备份数据所在的目录即可。

```
innobackupex --copy-back /path/BACKUP_DIR/2016-12-05_09-32-16
```

innobackupex 将所有的数据相关文件复制到服务器中的 datadir 目录，该目录由 my.cnf 文件中的 datadir 选项指定。最后的输出信息如下。

```
innobackupex: Finished copying back files.
161205 09:50:33 innobackupex: completed OK!
```

复制完成后，文件属性不会改变。在大多数情况下，在启动 MySQL 数据库之前，需要修改文件的所有权。

```
chown -R mysql:mysql /var/lib/mysql
```



注意：

- 在恢复的过程中，datadir 目录必须为空。如果不为空，innobackupex --copy-back 将不会复制，除非指定了--force-non-empty-directories 选项；
- 在恢复过程中，数据库需要处于关闭的状态（导入部分备份除外）。

增量备份

在日常的数据库备份中，并不是每次备份之间的数据都有所改变。DBA 可以只备份那些修改的部分。增量备份之所以可以做到这一点，是因为在 InnoDB 页面中有一个日志序列号 (LSN)，它充当数据库版本号的作用。每次修改数据库，这个数字就会增加。增量备份复制指定 LSN 位置以后的所有页面。

增量备份的优点如下。

- 节约备份所需的磁盘空间。
- 节约备份时间。

增量备份的缺点如下。

- 恢复操作相对较多。
- 恢复过程相对较慢, 需要基于全量备份或增量备份恢复。
- 不安全, 如果最原始的全量备份或上一个增量备份出现问题, 那么该增量备份就会出现
问题。

备份阶段

增量备份基于全量备份, 所以需要创建一个全量备份。具体操作, 前面已经做过讲述。在全量备份后, 备份目录中会存在 `xtrabackup-checkpoints` 文件。内容大致如下。

```

❏ backup_type = full-backupped
   from_lsn = 0
   to_lsn = 1626060
   last_lsn = 1626067
   compact = 0

```

增量备份需要指定 `--incremental` 选项和 `BASEDIR` 信息。

```

❏ innobackupex --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS
   --incremental --incremental-basedir=/path/BASE_BACKUP_DIR
   /path/INCREMENTAL_BACKUP_DIR/

```

增量备份完成后, 会在 `/path/INCREMENTAL_BACKUP_DIR/` 目录中创建一个时间目录, 例如: `/path/INCREMENTAL_BACKUP_DIR/2016-12-05_11-38-30`。同样, 在该目录中存在 `xtrabackup-checkpoints` 文件, 文件内容如下。

```

❏ backup_type = incremental
   from_lsn = 1626060
   to_lsn = 1626074
   last_lsn = 1626081
   compact = 0

```

增量备份还可以基于以前的增量备份完成数据备份。

```

❏ innobackupex --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS
   --incremental --incremental-basedir=/path/BASE_BACKUP_DIR
   /path/INCREMENTAL_BACKUP_DIR/

```

`xtrabackup-checkpoints` 文件内容如下。



```

backup_type = incremental
from_lsn = 1626074
to_lsn = 1626088
last_lsn = 1626095
compact = 0

```

注意：像前面所说，增量备份是基于 LSN 来完成的，所以增量备份也可以指定 LSN 值即可。例如：



```

innobackup --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS
--incremental --incremental-lsn=1626060 /path/INCREMENTAL_BACKUP_DIR/

```

- 增量备份只会影响 XtraDB 和 InnoDB 表，其他引擎将会复制所有的数据。

Prepare 阶段

增量备份的 prepare 阶段，与全量备份略有不同，需要注意的事项更多一些，如下。

- 首先，所有已提交的事务必须在每个备份上重放，这将完全合并到增量备份。
- 然后，为了得到可用的备份，未提交的事务必须回滚。

如果已经在全量备份上重放了提交的事务并回滚了未提交的事务，则将无法在此备份上添加增量。如果在增量备份上执行上述操作，则将无法添加其余的增量备份。

基于上述几点，prepare 阶段变得非常简单，指定 --redo-only 选项即可，如下。

- 在全量备份上执行 `innobackup --apply-log --redo-only BASE_DIR`。
- 在所有的增量备份上（除了最后一个增量备份）执行 `innobackup --apply-log --redo-only --incremental-dir=/path/INCREMENTAL_BACKUP_DIR BASE_DIR`。
- 在最后一个增量备份上执行 `innobackup --apply-log --incremental-dir=/path/INCREMENTAL_BACKUP_DIR BASE_DIR`。



注意：当合并增量备份时，--redo-only 应用于除了最后一个的所有增量备份。这就是为什么上面的第三步没有添加 --redo-only 选项的原因。如果在最后一步添加了该选项，则在恢复时，服务器需要执行回滚操作。

恢复阶段

通过 prepare 阶段，base 目录包含了所有数据。在恢复时，只需要通过以下操作即可。

```
innobackupex --copy-back /path/BASE_DIR
```

并行备份

当进行本地备份或使用 `xbstream` 选项备份的时候, 可以通过指定 `--parallel` 选项同时复制多个文件。此选项指定 `xtrabackup` 复制数据文件所创建的线程数。使用该选项的前提是必须启动 `innodb_file_per_table` 或利用 `innodb_data_file_path` 将共享表空间存储在多个文件中。虽然可以通过该参数加快备份速度, 但是由于此功能是基于文件级实现的, 所以在大量随机读请求重叠及并发文件传输时也会增加备份时 I/O。所以, 在使用此功能时, 需要考虑到系统的负载问题。如果数据保存在单个文件中, 该选项将不起作用。

```
innobackupex --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS
--parallel=10 /path/BACKUP_DIR
```

并行复制的优点: 加速备份过程中的复制速度, 缩短备份时间。

并行复制的缺点: 对磁盘 I/O 消耗较大。

其他备份

远程限速流备份

DBA 在日常运维中经常遇到两个场景: 1. 搭建一个从实例; 2. 本地磁盘空间不足, 然而又需要备份。这时, 远程备份可以非常好地满足需求。不过在远程备份时, 需要考虑到网卡的能力, 不能因为备份影响了线上业务的运行。这时可以考虑使用双网卡, 一块用来备份, 一块用来提供线上服务。如果没有这个条件, 可以考虑在备份时限制速度来达到目的。

例如需要将 A 机器的数据库备份到 B 机器上, 在 B 机器上开启一个端口利用 `nc` 和 `pv` 达到限速的目的, 如下。

1. `mkdir -p /tmp/path`
2. `nc -l port | pv -q -L 50m | tar -xi`

在 A 机器上执行备份, 主要端口需要对应上, 如下。

1. `mkdir -p /tmp/tmpdir;`
2. `innobackupex --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS`
`--stream=tar --tmpdir=/tmp/tmpdir --salve-info /tmp/path`
`/* (这个路径要与接收端B上nc的当前路径保持一致) | nc -nv ip(B机器的IP地址)`
`port(与B机器开启的端口一致) */`

远程限速流备份的优点如下。

- 因为采用流的方式, 不消耗本地磁盘空间。

- 通过该备份方式备份完成后, 数据库的最后位置 (binlog 位点) 就是最新的, 因为传输完成的时候, 备份才结束。相对别的备份方式, 该备份方式的 binlog 位点信息更新, 这样后期在搭建主从时可以快速同步。

远程限速流备份的缺点如下。

- 传输过程中, 如果网络出现问题, 那么备份将会失败。
- 由于限速, 导致备份时间拖长。



注意: 在限速方面, 可以通过 pv 方式, 但是 innobackupex 本身也提供了参数 `--throttle`, 限制磁盘的 I/O, 减少磁盘的压力。不过它仅仅限制对 InnoDB 的日志和文件的操作, 对其他的存储引擎没有效果。

复制环境备份

在线上环境中, 如果存在集群负载过大、从库需要维护等问题, 可能需要在主库下搭建一个新的从库, 但在主库中备份可能会使主库压力过大, 那么只能在从库上备份, 这时可以指定 `--slave-info` 选项完成。该选项将主服务器的二进制文件信息保存到 `xtrabackup_slave_info` 文件种, 当搭建主从时, 可以利用这个文件内容直接生成 `CHANGE MASTER` 命令。

在从库中备份时, `--safe-slave-backup` 选项可以与 `--slave-info` 选项结合使用, 备份命令如下。

```
innobackup --host=127.0.0.1 --port=3306 --user=DB_BACKUP --password=BACKUPPASS
--stream=xbstream --slave-info ./ > full_backup.xbstream
```



注意: 线上环境请谨慎使用参数 `--safe-slave-backup`。

案例实践与心得

有时候在备份时, 会因为报错 “Too many open files” 而失败, 一般情况下, 这是因为 MySQL 实例中创建的表太多导致的。但表太多是不是就不能备份了呢? 当然不应该是, 先来看一下相关报错信息, 如下。



```
/* ...省略... */
xtrabackup: using O_DIRECT
>> log scanned up to (57492900900309)
xtrabackup: Generating a list of tablespaces
```

```
>> log scanned up to (57492900918977)
>> log scanned up to (57492900949707)
>> log scanned up to (57492901090175)
2017-01-04 10:56:47 7f6cc32e0720 InnoDB: Operating system error number 24 in a file
operation.
InnoDB: Error number 24 means 'Too many open files'.
/* ...省略... */
```

为什么会报这个错误?

先分析日志 “xtrabackup: Generating a list of tablespaces”, innobackupex 首先准备获取表空间的一个列表。这时, 先看一下其源码, 如下。

```
msg("xtrabackup: Generating a list of tablespaces\n");

err = xb_load_single_table_tablespaces(xb_check_if_open_tablespace);
if (err != DB_SUCCESS) {
    return(err);
}
```

以上说明 innobackupex 在备份的时候, 加载了表空间, 继续看代码:

```
At the server startup, if we need crash recovery, scans the database
directories under the MySQL datadir, looking for .ibd files. Those files are
single-table tablespaces. We need to know the space id in each of them so that
we know into which file we should look to check the contents of a page stored
in the doublewrite buffer, also to know where to apply log records where the
space id is != 0.
@return DB_SUCCESS or error number */
UNIV_INTERN
dberr_t
xb_load_single_table_tablespaces(bool (*pred)(const char*, const char*))
/*=====*/
{
    ...
}
```

通过方法的注释可以看出, innobackupex 在备份的时候, 需要拿到每个 .ibd 文件的 space id 号, 便于检查存储在 doublewrite 缓冲区中页面的内容。所以 innobackupex 需要打开每个表文件, 而且不会关闭, 保持文件的开启是为了正确地处理 DDL 操作。虽然有备份选项 “--close-files”, 可以关闭开启的文件, 但是存在一定的风险, 所以不建议使用该选项。那么再看错误 “InnoDB: Operating system error number 24 in a file operation” 会发现, 这里调用了 InnoDB 来打开表文件, 那么问题来了, 打开多少表会导致 “Too many open files” 问题呢? 这

时候可以把问题定位到 MySQL 层面了。MySQL 影响打开文件限制的参数是 `open_files_limit`, 可以先了解一下这个参数的来龙去脉, 如下。

- 操作系统运行 `mysqld` 打开的文件数。此变量在允许时的值是操作允许的实际值, 并且可能不同于服务器启动时指定的值。在 MySQL 无法更改打开文件数的系统上, 该值为 0。
- `open_file_limit` 值基于系统启动时指定的值, 以及 `max_connections` 和 `table_open_cache` 的值。公式如下。
 - $\text{value1} = 10 + \text{max_connections} + (\text{table_open_cache}) * 2$
 - $\text{value2} = \text{max_connections} * 5$
 - $\text{value3} = \text{open_files_limit}$ 的值
- 服务器在启动的时候, 会使用这三个值中的最大值来作为获取文件描述符的数量。



备注: 如果 `open_files_limit` 未设置, 则该参数值为 `max_connections*5` 和 `ulimit -n` 中的较大值。

- 这时可以调整 `open_files_limit` 参数的大小, 然后重启数据库, 使它满足需求即可。
- 另外一个问题, 为什么不能直接通过修改操作系统的 `ulimit -n` 的值, 使得 MySQL 能够打开的文件数多一些呢? 通过上面的备注知道, 如果未设置 `open_files_limit`, 则可以通过 `ulimit -n` 来完成设置, 不过也需要重启数据库, 使配置生效。如果 `open_files_limit` 参数设置了, 那么修改操作系统的 `ulimit -n` 也是不起作用的。

建议与提醒

- 请使用高版本的 `innobackupex`。
- 最好都是 InnoDB 表, 如果有 MyISAM 表也可以, 现在这块已经优化改进了。
- 同机的不同实例最好分开备份。同时备份的话, 磁盘 IO 消耗大, 建议远程备份。
- 如果使用流备份到远程, 请控制好网卡和流量, 比如使用双网卡和限速。
- 尽量不要使用 `innobackupex` 自带的增量备份, 因为后期使用或恢复上不太方便, 因为要知道增量的策略, 做不好比较麻烦。
- 尽量不要使用 `innobackupex` 的部分备份, 恢复时比较麻烦, 而且容易出错, 因为要 import 表空间。
- 备份结束时, 请立即 `apply-log`, 这样能够知道备份集是否可用, 如果出现问题, 还可以立即再备份一次或报告出来。
- 备份过程中尽量不要有长查询, 因为备份有些操作要等长查询结束, 当然也可以通过设置参数来 kill 掉长查询。

MySQL 分库分表

分库分表是 MySQL 永恒的话题。一般情况下，MySQL 被认为是个简单的数据库，在数据量大到一定程度之后处理查询的效率会降低，如果需要继续保持高性能运转的话，就必须分库或分表了。关于数据量达到多大才到极限这一点，本章先不讨论。研究源码的同学已经证实，MySQL 或 InnoDB 内部锁粒度太大的问题，大大限制了 MySQL 提供 QPS 的能力或处理大规模数据的能力。在这一点上，一般的使用者只好坐等官方不断地推出优化版本。

分库分表的种类

首先说明，这里所说的分库分表是指把数据库中数据物理地拆分到多个实例或多台机器上去，而不是 MySQL 原生的 Partitioning。

这里稍微提一下 Partitioning，这是 MySQL 官方版本支持的，在本地针对表的分区进行操作，它可以将一张表的数据分别存储为多个文件。如果在写 SQL 的时候，遵从了分区规则，就能把原本需要遍历全表的工作转变为只需要遍历表里某一个或某些分区的工作。这样降低了查询对服务器的压力，提升了查询效率。如果分区表使用得当，也可能大规模地提升 MySQL 的服务能力。

但是这种分区方式，一方面，在使用的时候必须遵从分区规则写 SQL 语句，如果不符合分区规则，性能反而会非常低下；另一方面，Partitioning 的结果受到 MySQL 实例，或者说 MySQL 单实例的数据文件无法分布式存储的限制，不管怎么分区，所有的数据还是都在一个服务

器上,没办法通过水平扩展物理服务的方法把压力分摊出去。所以,限于篇幅,在讲分库分表时就不谈 Partitioning 这种分区方式了。

在 MySQL 运维实践中,拆分一般分为垂直拆分和水平拆分。

- 垂直拆分

在考虑数据库拆分的时候,一般情况下,应该先考虑垂直拆分。垂直可以理解为分出来的库表结构是互相独立各不相同的。

- 如果有多个业务,每个业务直接关联性不大,那么就可以把每个业务拆分为单独的实例、库或表。
- 如果在一个实例上,有多个数据库,那么从分摊压力的角度考虑,可以把每个数据库拆分到单独的实例上。
- 如果在一个库里面有多张表,那么可以把每张表拆分到不同的实例上。
- 如果你有一张表,但这个表里的字段很多,每个字段都有不同的含义,例如 user 表里面有姓名、生日、地址等个人信息,那么当改表太大的时候,就可以把每个字段独立出来拆分为一张新表。例如 user_birth 表,可以只有两个字段: user_id、user_birthday。

- 水平拆分

水平拆分是针对一张表来说的。在经过垂直拆分之后,如果数据量仍然过大,例如注册用户已经超过了 10 亿,那只好通过某种算法进行水平拆分。拆分后的结果是多张具有相同表结构的表,每张表里面存储一部分数据。拆分的算法依据很多,例如通过 id 取模 100、1024 的方式,以及通过区分不同日期的方式等,如图 24.1 所示。blog 表通过取模 100,分成了 100 份。log 表通过每个月一个表进行按时间的水平拆分。feed 表通过除以约定的值进行了水平拆分。

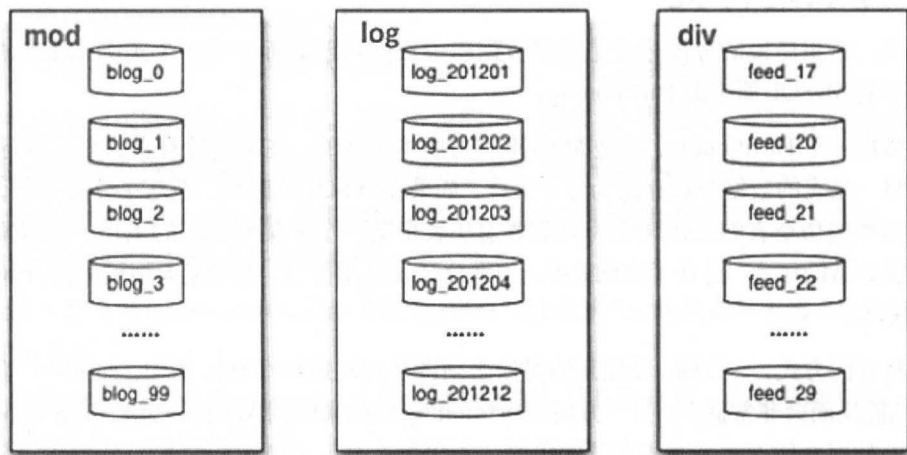


图 24.1

此外，还有一种拆分方式是水平拆分之后的垂直拆分。

进行拆分的目的是，通过这样的做法降低 MySQL 的负载，把原本不支持分布式存储的 MySQL 实例转换为基于 MySQL 的分布式集群。在进行水平拆分之后，如果数据库压力还是非常大，那么可以把水平拆分的结果再进行垂直拆分。

例如上面的 blog 表。先通过水平拆分把一张 blog 表拆分为 100 份；然后，可以根据业务的需求，再通过垂直拆分，把一部分水平拆分后的 blog_N 表迁移出去。按照目前的设计，在极端情况下，可以把 blog 表分布在 100 台物理机器上，每台物理机器上承担 blog 表访问压力的百分之一。

这样的操作在 MySQL Partitioning 中是无法完成的。

分库分表的原则

从一般运维的角度来看，什么情况下需要考虑分库分表呢？

原则零：能不分就不分。

是的，MySQL 是关系型数据库，数据库表之间的关系从一定的角度上映射了业务逻辑。任何分库分表的行为都会在某程度上提升业务逻辑的复杂度，数据库除了承载数据的存储和访问外，协助业务更好地实现需求和逻辑也是其重要工作之一。分库分表会带来数据的合并、查询或更新条件的分离，以及事务的分离等多种后果，业务实现的复杂程度往往会翻倍或指数级上升。所以，在分库分表之前，不要为分而分，而应该尽量去做其他力所能及的事情，例如升级硬盘、升级内存、升级 CPU、升级网络、升级数据库版本、读写分离及负载均衡等。所有分库分表的前提是，这些已经尽力做好了。

原则一：数据量太大，正常的运维影响正常的业务访问。

这里说的运维，包括如下三种情况。

- 对数据库的备份。如果单表或单个实例太大，在做备份的时候就需要大量的磁盘 IO 或网络 IO 资源。例如 1T 的数据，网络传输占用 50MB 的时候，需要 20000 多秒才能传输完毕，整个过程中的维护风险都是高于平时的。去哪儿网的做法是，给所有的数据库机器添加第二块网卡，用来做备份，或者 SST、Group Communication 等各种内部的数据传输。1T 的数据备份，也会占用大量的磁盘 IO，如果是 SSD 还好。当然，这里忽略某些厂商的产品在集中 IO 的时候会出一些 BUG 的问题。如果是普通的物理磁盘，则在不限流的情况下去执行 XtraBackup，但对于该实例来说基本不可用。
- 对数据表的修改。如果某个表过大，对此表做 DDL 的时候，MySQL 会锁住全表，这个时间可能会很长，在这段时间内业务不能访问此表，影响甚大。解决的办法有类似某些大厂 DBA 自己改造的可以在线秒改表的方法，不过他们目前也只是能添加字段而已，对

其他 DDL 还是无效；或者使用 `pt-online-schema-change`，当然在使用过程中，它需要建立触发器和影子表，同时也需要很长很长的时间，在此操作过程中的所有时间，都可以看作风险时间。把数据表切分，使总量减小，有助于改善这种风险。

- 整个表的热点数据，数据访问和更新频繁，经常有锁等待。例如曾经有个表 `user_last_login_time`，从名字可以看出来是记录用户最后一次登录时间信息的表。由于业务开展时的数据量比较小，并且考虑到每个注册用户只有一条记录，就没有分表。但是随着业务爆发性增长，这张表的数据量很快达到了 10 亿。这时候，DAU 在不到 1 亿的情况下，就有每天几个亿的 `update` 操作（同一个用户可能每天登陆好多次），虽然大部分都是简单的 `update` 操作，但由于更新太过频繁，也导致此表压力很大，经常出问题。

这个时候，又没有能力去修改源码，降低锁的粒度，那么只会把其中的数据物理拆开，用空间换时间，变相降低访问压力。这个案例中的热点表拆分，用到了水平拆分。

原则二：表设计不合理，需要对某些字段进行垂直拆分。

这里举一个例子，如果有一个名为 `users` 用户表，在最初设计的时候可能是如下这样的。

id	bigint	用户的 ID
name	varchar	用户的名字
last_login_time	datetime	最近登录时间
personal_info	text	私人信息
xxxx		其他信息字段

一般的 `users` 表会有很多字段，这里不再列举。如上表所示，在一个简单的应用中，这种设计是很常见的。但是，设想一下以下两种情况。

- 设想情况一：你的业务中彩了，用户数从 100 万飙升到 10 个亿。为了统计活跃用户，在每个人登录的时候都会记录一下他最近登录的时间。并且有的用户活跃得很，不断地去更新这个 `login_time`，使这个表不断地被 `update`，压力非常大。那么，在这个时候，只要考虑对它进行拆分，站在业务的角度，最好的办法是先把 `last_login_time` 拆分出去，并将拆分出来的字段命名为 `user_last_login_time`。这样，业务的代码只有在用到这个字段的时候修改一下就好了。如果不这么做，直接把 `users` 表水平切分，那么所有访问 `users` 表的地方都要修改。或许你会说：“我有 `proxy`，能够动态 `merge` 数据。”到目前为止，我还从没看到过 `proxy` 不影响性能的情况。
- 设想情况二：`personal_info` 这个字段本来没什么用，就是让用户注册的时候填一些个人爱好而已，基本不会对它做查询。一开始的时候，有它没它无所谓。但是后来发现了两个问题，①这个字段占用了大量的空间，因为是 `TEXT`，所以有很多人喜欢在这里长篇

大论地介绍自己；②更糟糕的是，不知道哪天哪个产品经理心血来潮，说允许个人信息公开吧，以便让大家更好地相互了解。那么，在所有人猎奇窥私心理的影响下，对此字段的访问量就会大幅度增加。这时，数据库压力瞬间抗不住了，只好考虑对这个表进行垂直拆分。也就是，把这个字段加上用户的 id 单独独立出去。关于 TEXT 的拆分，这里多说一句，如果一个查询表里存在 TEXT 或 BLOB 字段，而且这个查询需要创建内部临时表的话，那它不能使用内存临时表，必须使用磁盘临时表，不然会对性能造成巨大影响，也会占用物理磁盘大量 IO，最终导致性能剧烈下降。一般情况下，这是必须要拆分的情形。

原则三：某些数据表出现了无穷增长的情况

例子很好举，各种的评论、消息、日志记录。这个增长不是跟人口成比例的，而是不可控的，例如社交网站业务的 feed 广播，发一条消息会扩散给很多很多人。虽然主体可能只存一份，但不排除一些索引或路由有这种存储需求。这个时候，通过增加存储提升机器配置已经显得苍白无力，水平切分是最佳实践。拆分的标准很多，按用户的、按时间的、按用途的，不再一一举例。

原则四：安全性和可用性的考虑

这也算是拆分带来的意外惊喜吧。不要把所有鸡蛋放在一个篮子里，我们不希望数据库出问题，但出不出问题不是我们能决定的，或者可以这么说，出问题几乎是一定的。我能决定什么呢？我希望在出问题的时候不要影响到 100% 的用户，影响的比例越小越好，那么水平切分可以解决这个问题。把用户、库存、订单等本来统一的资源切分掉，每个小的数据库实例承担一小部分业务，这样整体的可用性就会提升。这对 Qunar 这样的业务还是比较合适的，人与人之间、某些库存与库存之间关联不太大的时候，可以做一些这样的切分。

原则五：业务耦合性考虑

这一点跟原则四有些类似，主要是站在业务的层面上来看。火车票业务和烤羊腿业务是完全无关的业务，虽然每个业务的数据量可能不太大，放在一个 MySQL 实例中完全没问题，但是烤羊腿业务的 DBA 或开发人员水平很可能很差，动不动就给你出一些幺蛾子，直接把数据库搞挂。这时，虽然火车票业务的人员技术很优秀，工作也很努力，但照样被老板打屁股。解决的办法很简单：惹不起，躲得起！

《三国演义》第一回：“话说天下大势，分久必合，合久必分。”其实在实践中，有时候可能原本要分，后来又发现分了还得合，分分合合，完全是现实的需求，随需而变才是王道，而 DBA 的价值也能在此体现。或分或合的情况太多，不能穷举。

分库分表实现

分库分表的实现,主要体现在两个方面。一方面是把数据库原来数据的存储位置迁移到新的库表里;另一方面,数据库的拆分实际上破坏了数据库的关系逻辑,原本的 SQL 访问可能不再适用,在业务层,怎样才能最好地继续使用数据库,也是实现的难点。

数据库层的实现

在数据库层面,DBA 可以根据以上所述的分库分表的种类和原则,通过必要的迁移实现数据的拆分。在实践中,一般有几办法实现。当然,这里讨论的都是在不不停机的情况下完成 0 宕机,或者在尽可能小的宕机时间里进行平滑迁移,这里不讨论把整个数据库或业务停掉,然后 dump、load 数据的情况。

- 针对不改变表结构的库表拆分,可以通过 MySQL Replication 或做不同的 Slave 从库去实现。在 MySQL Replication 的配置中,有以下几个参数值得注意。
 - Replicate_Do_DB: SQL Thread 只同步某一个数据库 Binlog。
 - Replicate_Ignore_DB: SQL Thread 忽略某个数据库的 Binlog。
 - Replicate_Do_Table: SQL Thread 只同步某个表的 Binlog。
 - Replicate_Ignore_Table: SQL Thread 忽略某个表的 Binlog。
 - Replicate_Wild_Do_Table: SQL Thread 通配匹配某些数据库的 Binlog。
 - Replicate_Wild_Ignore_Table: SQL Thread 通配忽略某些数据库的 Binlog。

可以看出,通过以上几个参数的组合,可以很轻松地限定新做的 MySQL Slave 只保留哪些库表。这些就是拆分之后的结果。

- 针对需要改变表结构的库表进行拆分,例如表的垂直拆分或水平拆分。

第一种做法是 DBA 自己实现。首先,DBA 需要把新的表结构建好,跟老的表结构在同一个实例下面。其次,DBA 写触发器,针对表的增删改操作,用触发器把数据同步到新表中,同时在后台同步历史数据库,以完成数据表的全部同步。这个做法有点像著名的 pt-online-schema-change 的原理。

由于有时候要切分为很多很多的表,所以需要很多的触发器,曾经写过一个生成触发器的生成模板,这里举例如下。



```
#!/bin/sh
```

```
echo "DELIMITER \|" echo "drop trigger if exists ins\_gift; create trigger
ins\_gift after insert on gift\_record for each row" echo "begin" echo " set @v
= 0;" echo "set @v=NEW.receiver\_uid%100;" echo " case @v" for i in {0..99}; do
echo " when $$i then " echo " insert into gift\_record\_receiver\_$$i set
```

```

id=NEW.id,gid=NEW.gid,gname=NEW.gname,gimage=NEW.gimage,gprice=NEW.gprice,sender
\_uid=NEW.sender\_uid,sender\_name=NEW.sender\_name,receiver\_uid=NEW.receiver\_
uid,receiver\_name=NEW.receiver\_name,method=NEW.method,message=NEW.message,addt
ime=NEW.addtime,status=NEW.status,flash\_data=NEW.flash\_data,kind=NEW.kind,payw
ay=NEW.payway,is\_wap=NEW.is\_wap;"
done; echo " end case;"

```

```

echo "set @v=NEW.sender\_uid%100;" echo "case @v" for i in {0..99}; do echo "
when \${i} then "

```

```

echo " insert into gift\_record\_sender\_ \${i} set
id=NEW.id,gid=NEW.gid,gname=NEW.gname,gimage=NEW.gimage,gprice=NEW.gprice,sender
\_uid=NEW.sender\_uid,sender\_name=NEW.sender\_name,receiver\_uid=NEW.receiver\_
uid,receiver\_name=NEW.receiver\_name,method=NEW.method,message=NEW.message,addt
ime=NEW.addtime,status=NEW.status,flash\_data=NEW.flash\_data,kind=NEW.kind,payw
ay=NEW.payway,is\_wap=NEW.is\_wap;"

```

```

done; echo " end case;" echo "end;"

```

```

echo "drop trigger if exists update\_gift; create trigger update\_gift after
update on gift\_record for each row" echo "begin" echo " set @v =0;" echo " set
@v=NEW.receiver\_uid%100;" echo "case @v" for i in {0..99}; do echo " when \${i}
then " echo " update gift\_record\_receiver\_ \${i} set
status=NEW.status,method=NEW.method where id=NEW.id;" done; echo " end case;"

```

```

echo " set @v=NEW.sender\_uid%100;" echo "case @v" for i in {0..99}; do echo "
when \${i} then " echo " update gift\_record\_sender\_ \${i} set
status=NEW.status,method=NEW.method where id=NEW.id;"

```

```

done; echo " end case;" echo "end;"

```

```

echo "drop trigger if exists delete\_gift;create trigger delete\_gift before
delete on gift\_record for each row" echo "begin" echo " set @v = 0;" echo " set
@v=OLD.receiver\_uid%100;" echo "case @v" for i in {0..99}; do echo " when \${i}
then " echo " delete from gift\_record\_receiver\_ \${i} where id=OLD.id; " done;
echo " end case;"

```

```

echo " set @v=OLD.receiver\_uid%100;" echo "case @v" for i in {0..99}; do echo "
when \${i} then " echo " delete from gift\_record\_sender\_ \${i} where id=OLD.id; "

```

```

done; echo " end case;" echo "end ;" echo "DELIMITER ;"

```

由于在 MySQL 里面动态创建触发器这样的操作具有很大的风险性,同时触发器本身也会影响 MySQL 的性能。所以,后来考虑能否也按照上面提到的 MySQL Replication 的方式去实现相应的对表结构的垂直拆分。但是,原生的 MySQL Replication 限制太多,无法实现。因此我们的思路是,通过模仿 MySQL Replication 的原理,实时地同步 Binlog 数据,但这些数据经过转换,可以自己自由修改,这样就可以很轻松地做异构数据的同步,从而完成对数据表的拆分了,这也是 Inception Gate 需求的原型。

业务层的实现

在业务层面,怎么去读写经过拆分之后分布式存储的数据,这也是分库分表需要解决的。一般的实现方式有以下两个方向。

- 第一,通过在业务程序端嵌入客户端软件包的形式,在业务需要访问数据库的时候,首先访问路由表,再由路由表判断需要访问 MySQL 数据库的位置,然后业务端直接连接 MySQL 数据库,做直接的 SQL 操作。如果需要进行数据的合并,则在业务程序端实现,或者在嵌入的客户端软件包里面实现。这种方式的优点是,业务端程序是直接连接数据库的,没有任何不兼容的影响,也没有任何性能方面的影响。但是,业务端需要合并数据,逻辑复杂,特别是需要业务端嵌入额外的软件包,不具有通用性。但在公司内部使用,却是非常灵活高效的。
- 第二,通过 Proxy 的形式,提供 RDS 的服务。这是目前大部分云服务的工作方式。Proxy 实现的优劣会直接影响服务的兼容性和效率。目前来看,不管实现多么优秀的 Proxy,由于业务端发送的 SQL 总是要经过一层的转发,所以总会存在兼容性和性能问题。目前在开源社区里,有非常多针对 MySQL 的 Proxy 产品,这里就不再一一赘述了。

随着业界对分库分表的需求越来越大,MySQL 官方在经历了长期 MySQL Proxy 项目的尝试之后,最近也推出了新的解决方案,那就是 MySQL Connector+MySQL Router+MySQL Fabric 的解决方案。这是一个很大的话题,再加上到目前为止,这个方案还在不断优化的过程中,所以会放到以后再详细讲述它们的原理、用法和使用心得。

25 MySQL 数据安全

一直以来，数据都被视为核心资产，互联网行业如此，各行各业亦是如此。特别是现在大数据概念深入人心，大数据应用层出不穷，人们对数据价值的认识也日益加深。作为存储数据最重要的载体，关系型数据库在整个互联网或大数据产业链中扮演的角色尤为重要，数据存储的安全与否，是人们普遍关心的重点问题。一般的观点认为，在进行数据库选型的时候，收费的商业数据库例如 MS SQL Server、Oracle Database 等，有比较好的数据安全保障，可以放心使用，所以它们在传统的金融、政府、企业中的应用相当普遍；而新兴的开源数据库，例如最流行的 MySQL，给人的感觉是可以在互联网这样的场景中使用，反正互联网的数据没有类似银行这样的业务数据重要，丢了就丢了吧，影响不大，在企业中使用则让人不能放心，这是对 MySQL 认识不够深入而产生的误解。在本章中，集中介绍 MySQL 是如何来保障数据安全的，以打消人们使用过程中在这方面的心理障碍。



注意：这里说的数据安全是指数据库存储数据的解决方案，针对的是机器硬件不可用之类的情况，对于网络安全、黑客攻击之类的安全问题，我们另行讨论。

MySQL 在保障数据安全存储方面，提供了很多可配置的功能和特性，正确使用这些配置和特性，是保障数据安全的根本。可以从以下角度来阐述。

单机安全

从单机或 MySQL 单实例的角度来看,数据落盘的安全主要考虑 MySQL 实例进程挂掉和 MySQL 实例所在的操作系统挂掉的情况。类似于其他关系型数据库,MySQL 提供了 Undo log 和 Redo Log 来保证数据库非正常重启后数据恢复的机制。同时,MySQL 采用了特有的 Double Write 机制来保证写入磁盘数据的完整性和正确性,Double Write 可以用参数 `innodb_doublewrite` 来控制。而日志刷盘的策略,有个非常重要的参数叫 `innodb_flush_log_at_trx_commit`,这个参数取值的不同决定了日志刷盘策略的不同,也决定了 MySQL 提供的数据安全级别的不同。

集群安全

在任何时候,都可能会出现数据库服务器硬件或操作系统故障,继而导致物理服务器无法启动,操作系统不能正常加载,或者是 MySQL 进程无法启动的情况。这时候,要保障 MySQL 的数据安全,需要借助 MySQL 集群部署,并根据 MySQL 的特性正确配置。在使用 MySQL 集群的过程中,又有两种截然不同的处理思路。

- 使用 MySQL 原生的 Replication 来搭建基于 Master-Slave 拓扑结构的异步 MySQL 集群。MySQL 开发者很早就意识到了数据库复制的重要性,在 2000 年左右就已经开始支持 Replication。随着 MySQL 功能的不断健全,MySQL 对 Replication 的支持也越来越完善。Binlog 的存储优化及可控的刷盘策略、Replication 信息存储、半同步机制、多线程复制技术、多源复制技术等,最终都优化了 MySQL 集群的搭建,为 MySQL 数据安全提供了保障。
- 使用 Galera 这样的第三方插件,或者不太成熟的 MySQL 官方的 Group Replication,搭建真正的同步复制 MySQL 集群。Galera 是目前最先进的 MySQL 高可用、高一致性解决方案,它为 MySQL 的运维提供了一条崭新的思路。我们在最近几年一直实践基于 Galera Cluster 的 MySQL 高可用方案,取得了巨大的成功。同时,MySQL 官方也看到了 Galera 的价值,正在开发自己的类似方案 Group Replication,这个方案现在已经 GA (General Availability),其前景非常光明,值得关注。

备份安全

如果碰到了极端情况,我们提供线上服务的 MySQL 主库宕机,同时这个集群的其他从库也不可用。那么,如果要保障数据安全,唯一可以依赖的就是数据库的备份了。所以,正确及时有效的数据库备份是必不可少的,MySQL 是在生产中久经验证的,其支持备份的各种工具和方案也层出不穷,目前比较流行的 MySQL 备份工具是由 Percona 出品的 Percona XtraBackup,

在第 23 章中已经讲述过 XtraBackup 的实现原理和使用细节，这里不再赘述。不过，要着重强调的是，数据库的备份不仅要注意备份的时效性和恢复时间最小化的问题，而且备份机的安全性和备份集的可用性也是需要重视的，这关系到备份数据是否真正可用。近年来一直有各种误删除数据的故障发生，而最终需要恢复数据的时候往往发现备份数据不可用，这是非常悲剧的。

MySQL 实例安全保证

在 MySQL 内部，有很多机制来保证数据库自身的数据安全，我们主要关注 InnoDB 引擎。本章将从 InnoDB 的日志管理机制原理说起，逐步深入，并且结合 MySQL 的各种参数组合对数据库安全方面的影响，把 MySQL+InnoDB 安全管理数据的机制说清楚。

Double Write

本书中已经讲述过 Double Write 的机制，细节这里不再叙述。它保证了数据页的正确性，虽然开启 Double Write 之后，每个页面都写了两遍，这对数据库性能有一定的影响，但是由于所写的页面都缓存到内存中，每一部分的缓存空间满了之后才真正写入到文件中，而且两次写入是将若干数量的页面组合起来形成连续的空间写入到两次写的空间中，这样就形成了顺序写，对于磁盘来说，顺序写消耗的性能比较低。实际上经过测试，两次写使性能降低了约 10%。当然，这是针对普通的机械磁盘而言的，对目前比较流行的 SSD 来说，两次写已经不是问题，性能影响可能更小。如果未开启此参数，可能导致页面写失败，那么 MySQL 在宕机时，可能导致数据库无法启动，丢失数据。具体原理细节详见第 10 章。

REDO LOG

REDO LOG 是用来做数据库 crash recovery 的，这是保证数据库数据安全非常重要的功能之一。Redo 日志记录了所有对 InnoDB 数据库的修改操作。在 InnoDB 存储引擎中，一般默认有两个日志文件，数据库创建之后，会自动新建两个名为 ib_logfile0、ib_logfile1 的文件。如果这两个文件不存在，则 InnoDB 在启动的过程中，会根据配置的参数或默认值，重新创建 REDO LOG 文件。

REDO LOG 刷盘机制

当提交事务（逻辑）时，可以通过参数 `innodb_flush_log_at_trx_commit` 来控制 REDO LOG 写入的机制，参数值不同，产生的行为不同。主要的参数值如下。

- `innodb_flush_log_at_trx_commit=0`：事务提交时，MySQL 不会去处理日志缓存区的内容，也不会去处理日志文件的刷盘操作，由 MySQL 的后台 Master 线程每隔 1s 将缓存区的文件刷新到日志文件中。

- `innodb_flush_log_at_trx_commit=1`: 事务提交时, 会将日志缓冲区的日志写入到文件中, 同时会刷新到磁盘中, 保证数据库事务完全不会丢失。这种设置影响数据库性能。
- `innodb_flush_log_at_trx_commit=2`: 事务提交时, 会将日志缓存区日志写入到文件中, 但是不会刷新到磁盘中。由 MySQL 的后台 Master 线程每隔 1s 将系统缓存的日志文件刷新到磁盘中。

关于参数设置的参数值与日志缓冲区、OS cache、日志文件 (ib_logfile) 之间的关系, 可以通过图 25.1 来表示, 其中左侧浅色背景为内存部分, 右侧深色背景为磁盘文件, 箭头表示日志数据的流向。

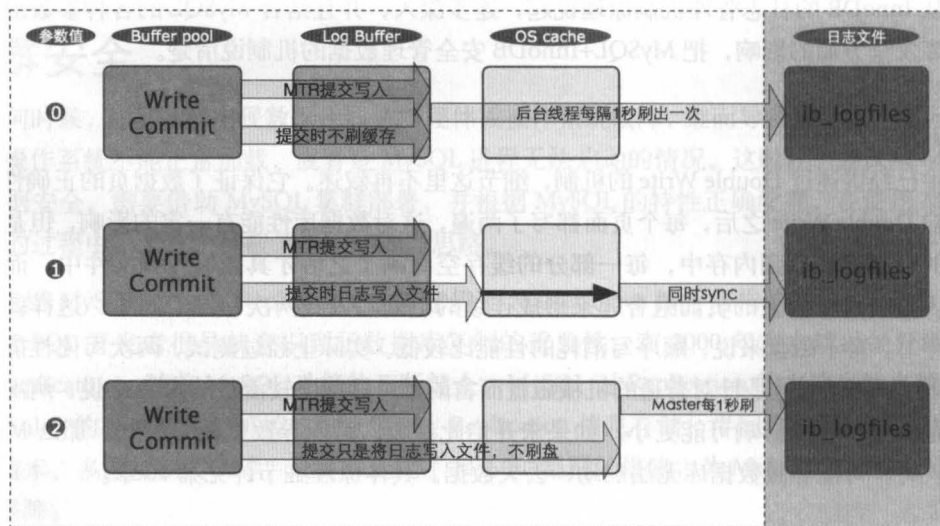


图 25.1

宕机与 `innodb_flush_log_at_trx_commit` 的关系

数据库宕机主要有以下两种情况。

- 数据库宕机, 但数据库所在的服务器还在正常运行。
- 数据库所在的服务器宕机。

关于数据丢失, 涉及几个问题。在宕机发生时, 已经提交完成的数据库修改, 能否保证 REDO 日志真正写入到磁盘上, 如果不能, 就被认为是数据丢失; 在宕机恢复后, 提交的数据库修改是否还存在, 如果不存在, 则被认为是数据丢失; 如果是事务提交了, 但提交失败了, 最终相关修改没有生效, 这样的情况, 就不能被认为是数据丢失了。接下来, 以“数据丢失量”为主线, 详细描述宕机方式与参数配置之间的关系。

1. 数据库宕机、服务器系统是正常运行。

- `innodb_flush_log_at_trx_commit=0`：由于事务提交时，不刷新日志缓存区，即使事务已经提交成功了，缓冲区的日志也会全部丢失，所以最新的数据修改也都会丢失。前面说过，Master 线程每秒刷新一次，所以一般只丢失最近 1s 的事务。
- `innodb_flush_log_at_trx_commit=1`：如果数据库宕机，由于每次事务提交都会刷新到磁盘中，所以数据不会有任何丢失。
- `innodb_flush_log_at_trx_commit=2`：如果数据库宕机，由于事务已经刷新到 OS cache 中，然而服务器系统并没有出现问题，这样日志还会被刷新到磁盘中，那么数据就不会丢失。

2. 数据库所在服务器宕机。

- `innodb_flush_log_at_trx_commit=0`：由于事务提交时，不刷新日志缓存区，即使事务已经提交成功了，缓冲区的日志也会全部丢失，所以最新的数据修改也都会丢失。前面说过，Master 线程每秒刷新一次，所以一般只丢失最近 1s 的事务。这种情况与数据库宕机是一样的道理，有相同的影响。
- `innodb_flush_log_at_trx_commit=1`：由于每次事务提交都会刷新到磁盘文件中，所以即使是服务器宕机，数据也不会有任何丢失。
- `innodb_flush_log_at_trx_commit=2`：由于事务只是刷新到 OS cache 中，如果服务器系统出现问题，导致宕机，那么日志就没有被刷新到磁盘中，就会丢失最近 1s 的事务。

关于参数设置的参数值、宕机方式与数据丢失之间的关系，可以通过下表来表示。

<code>innodb_flush_log_at_trx_commit</code>	数据库宕机	OS 宕机
0	丢失最多 1s 数据	丢失最多 1s 数据
1	不丢数据	不丢数据
2	不丢数据	丢失最多 1s 数据

MySQL 集群安全保证

一般在生产环境下，很少有 MySQL 单实例支撑业务的情况，大部分的 MySQL 应用都是采用搭建集群的方法，搭建 MySQL 集群，进行数据库层面的读写分离、负载均衡或数据备份。而 MySQL 集群在本质上可以分为两类，一类是依赖 MySQL Replication 的传统集群，另一类是依赖插件的真正的 MySQL 集群化解决方案。

传统的主从模式如何保证数据库安全

基于 MySQL 原生的 Replication 是最常见的保证数据库数据安全的机制，在数据库发生宕机后，其他节点还能快速提供服务，并且保证数据库的数据不丢失。那么，如何保障数据库安全呢？在主从模式中又有哪些参数影响数据库的数据安全？接下来，将会从参数的配置方面来描述主从模式是如何保障数据库安全的。主要通过参数 `sync_binlog`、主从复制相关参数（`master_info_repository`、`relay_log_info_repository`、`relay_log_recovery` 等）来叙述主从复制保证数据库安全的方式。

`sync_binlog`

Binlog 是用于保存数据库修改的日志信息。传统的主从复制是基于 Binlog 的，Binlog 的安全直接关系到主从复制的安全，而 Binlog 的写入方式主要由参数 `sync_binlog` 来控制。参数值决定了其行为方式，主要参数值如下。

- `sync_binlog=0`：事务提交时，MySQL 将 Binlog 信息写入 Binlog 文件（OS Cache）中，但是 MySQL 不控制 Binlog 的刷磁盘操作，由文件系统自己控制其缓存的刷新。这是最危险的，一旦操作系统宕机，在 Binlog cache 中的所有 Binlog 都会丢失。如果只是数据库宕机，而操作系统未宕机，那么数据库所生成的 Binlog 都不会丢失。
- `sync_binlog=1`：每一个事务提交时，MySQL 都会把 Binlog 刷新到磁盘中。这样，数据库安全性最高，但是性能损耗也是最大的。如果这样设置的话，在数据库或操作系统宕机的情况下，二进制日志中缺少的任何事务也只能处于准备阶段，那么导致服务器自动恢复时，会回滚这些事务，保证无数据丢失。虽然 Binlog 是顺序 IO，但是多个事务同时提交，同样会对 MySQL 和 IO 的性能带来很大影响，不过 MySQL 可以通过 Group Commit 来缓解这种压力。
- `sync_binlog=N N>1`：表示每 N 次事务提交，MySQL 调用文件系统的刷新操作将缓存刷新到磁盘中。如果数据库或操作系统在这个时候宕机，数据库可能会丢失一些事务。



注意：在 MySQL 5.7 版本中，由于添加了多线程复制，在写 Binlog 方面，也做了相应的调整，具体请参考本书中有关多线程复制原理的部分。

两阶段提交

MySQL 开启 Binlog 后，所有的事务都会产生 Binlog Event，这些 Event 会被看成事务数据的一部分。因此要保证事务的 Binlog Event 和 InnoDB 引擎中的数据一致性。所以，MySQL 宕机重启后需要保证如下四点。

- 所有已经提交事务的数据仍然存在。
- 所有没有提交事务的数据自动回滚。
- 所有已经提交事务的 Binlog Event 也仍然存在。
- 所有未提交事务没有记录 Binlog Event。

如果数据库重启后数据还在，但是 Binlog Event 没有了，就没有办法把这部分数据复制到其他节点上；如果重启后数据没有了，但是 Binlog Event 还在，那么不存在的数据就会复制到其他节点上。这都会导致主从数据不一致的情况。为了保证 Binlog 的宕机安全，MySQL 内部使用了两阶段提交。

MySQL 在开启 Binlog 后，MySQL 内部会自动将普通事务当作一个 XA 事务来处理，在提交事务的过程中，MySQL 会自动为每个事务分配一个唯一的 ID (XID)，XID 会被记录到 Binlog 和 InnoDB Redo log 中。事务在提交时会自动被分成 Prepare 和 Commit 两个阶段。

- Prepare 阶段：告诉 InnoDB 引擎做 Prepare，InnoDB 更改事务状态，并将 Redo log 刷入磁盘。
- Commit 阶段：先记录 Binlog 日志，然后告诉 InnoDB 引擎 Commit。

大致了解了两阶段提交，那么该机制是如何保证数据安全的呢？在数据库宕机重启后，事务可能处于以下四种状态。

- InnoDB 引擎已经提交 (Commit)。根据两阶段提交可知，Binlog 中一定记录了该事务的 Events，所有事务是一致的，无须处理。
- 在 InnoDB 中已经完成了 Prepare 阶段，Binlog 中已经有该事务的 Events，但是 InnoDB 引擎未提交。需要通知 InnoDB 引擎提交这些事务。
- 在 InnoDB 中已经完成了 Prepare 阶段，Binlog 中没有该事务的 Events。因为 Binlog 没有记录，需要通知 InnoDB 回滚这些事务。
- InnoDB 还未完成 Prepare 阶段。根据两阶段的过程，Binlog 也没有该事务的 Events，需要通知 InnoDB 回滚这些事务。

MySQL 通过两阶段提交的方式保证宕机安全，这需要 Server 层、Binlog 和 InnoDB 共同协作完成。

innodb_support_xa 参数

默认情况下，innodb_support_xa 是开启的，如果将该参数关闭，事务提交可以以不同的顺序写入 Binlog。如果宕机恢复或从库应用这些 Binlog，就可能导致数据的不同，这是非常不安全的。建议在线上环境或主库中开启。

`innodb_support_xa` 在 MySQL 5.7.10 中已经废除, 并且将在未来的 MySQL 版本中删除。这是因为在 MySQL 5.7.10 中 InnoDB 支持在 XA 事务中始终启用两阶段提交, 所以就已经没有必要再用这个参数了。

主从复制参数的影响

在 Replication 的配置中, 有一些参数会影响主从复制的行为, 这些行为的配置会影响复制的数据一致性问题, 进而影响到集群的数据安全。

`binlog_format`

`binlog_format` 主要有三种格式: STATEMENT、ROW 和 MIXED。

下面简单介绍不同的 Binlog 格式与复制之间的关系。

- `binlog_format = STATEMENT`: MASTER 写入执行的 SQL 语句到 Binlog 中, 从库读取这些 SQL 语句并且执行。这种基于语句的复制 (SBR) 是 MySQL 最早支持的形式。

基于 STATEMENT 格式 (SBR) 的复制存在一些优势, 如下。

- 技术成熟。
- 减少了 Binlog 的写入量。
- Binlog 包含了所有的修改语句, 便于审计。

但是也存在很多缺点如下。

- 基于 STATEMENT 的复制是不安全的, 有些函数不能正确地在 SLAVE 上复制。
- 与基于 ROW 格式的复制相比, `INSERT...SELECT` 需要更多的锁。
- 隔离级别必须要设置为 `Repeatable-Read`, 而这是发生死锁的元凶之一。
- `binlog_format = MIXED`: 可以将 MASTER 的 `binlog_format` 配置成同时使用基于 STATEMENT 和 ROW 两者的组合格式, 它记录日志取决于修改的类型, 选择最适合的格式来记录该修改。默认情况下, 使用 STATEMENT 格式记录日志, 在特定的情况下转换成基于 ROW 格式记录。
- `binlog_format = ROW`: MySQL 在 5.7.7 版本之后, 把 `binlog_format` 的默认值修改为了 ROW。MASTER 将修改表的 event 写入 Binlog 中, 并且 MASTER 将该 Binlog 信息发送到 SLAVE, SLAVE 重放 Binlog 中 event。基于 ROW 格式 (RBR) 复制是最安全的复制, SLAVE 需要的行锁更少。但是也有一些缺点, 那就是基于 ROW 格式的复制, Binlog 会记录更多的数据。并且无法在 SLAVE 上看到从 MASTER 上获取的语句, 因为都是 event。但是在 ROW 格式下, 可以开启 `binlog_rows_query_log_events` 参数, 这会让 Binlog 在记录 events 的同时, 也记录下原始的 SQL 语句, 以方便后续的查询或审计。

在复制过程中, 建议使用 ROW 格式的模式, 其他模式可能会导致主从数据不一致的情况。

master_info_repository 与 sync_master_info

其中, master_info_repository 有两个值, 分别是 FILE 和 TABLE, 该参数决定了 SLAVE 记录 MASTER 的状态, 如果参数值是 FILE, 就会创建 master.info 文件; 如果参数值是 TABLE, 就在 mysql 库中创建 slave_master_info 的表。



注意: 如果使用多源复制, 请将参数值设置为 TABLE。

其中, sync_master_info 参数决定 SLAVE 刷新 MASTER 的状态的方式。并且 master_info_repository 的参数不同, 刷新方式也不同。

- 如果 master_info_repository = FILE, sync_master_info = N, 其中 $N > 0$, 那么 SLAVE 就会在每 N 个事件后, 使用 fdatsysnc() 方式同步 MASTER 状态信息到 master.info 文件中。如果 sync_master_info = N, 其中 $N = 0$, 那么 MySQL 只会将状态信息写入 OS Cache 中, 需要等待操作系统同步。
- 如果 master_info_repository = TABLE, sync_master_info = N, 如果 $N > 0$, 那么 SLAVE 就会在每 N 个事件后, 更新 mysql.slave_master_info 表。如果 $N = 0$, 那么 mysql.slave_master_info 表将永远不会更新。

relay_log_info_repository 与 sync_relay_log_info

其中 relay_log_info_repository 用来决定 SLAVE 同步的位置信息记录在哪里, 同样有两个参数。如果 relay_log_info_repository = FILE, 就会创建一个 relay-log.info; 如果 relay_log_info_repository = TABLE, 就会创建 mysql.slave_relay_log_info 表用来记录同步的位置信息。



注意: 如果使用多源复制, 请将参数值设置为 TABLE。

其中, sync_relay_log_info 参数用来控制 SLAVE 同步位置的刷新方式, 受 relay_log_info_repository 参数影响。

- relay_log_info_repository = FILE, sync_relay_log_info = N, 如果 $N > 0$, SLAVE 在 N 个事务之后使用 fdatsync() 方式将 relay_log.info 文件同步到磁盘中。如果 $N = 0$, 那么 MySQL 并不会同步 relay_log.info 文件到磁盘, 而是让操作系统决定。
- relay_log_info_repository = TABLE, 如果该表是支持事务的表, 那么 SLAVE 在每次事务之后, 都会更新该表, 忽略 sync_relay_log_info 参数。如果该表是非事务表, sync_relay_log_info=N, 当 $N > 0$ 时, 则 SLAVE 会在每 N 个 event 后更新该表; 当 $N = 0$ 时, 则该表永远不会被更新。

relay_log_recovery

该参数默认是打开的,在数据库启动后立即启动自动 relay log 恢复。在恢复的过程中,创建一个新的 relay log 文件,将 SQL 线程的位置初始化到新的 relay log,并将 I/O 线程初始化到 SQL 线程的位置。

MySQL 在运行的过程中,从库上可能会出现意外宕机的情况,那么在从库启动后,必须能够恢复到停止之前的状态。I/O 线程必须恢复到已经接收事务的位置,SQL 线程必须恢复到已经执行事务的位置。该信息传统上是存储在文件中,那么就有可能存在不一致或损坏的风险。从 MySQL 5.7 开始,可以使用表来存储这些信息,并把这些表设置为 InnoDB 引擎,通过使用事务型存储引擎,总是能够恢复这个信息。可以配置参数 `master_info_repository = TABLE` 和 `relay_log_info_repository = TABLE` 使从库信息存储在表中。从库如何从宕机的状态恢复到正确的状态,取决于从库是单线程还是多线程、`relay_log_recover` 参数的值,以及 `MASTER_AUTO_POSITION` 的使用方式。

1. 单线程模式的复制 (MySQL 5.7)。

- 当基于 GTID 模式复制的时候,并且设置了 `MASTER_AUTO_POSITON` 参数和 `relay_log_recovery = 0`,使用该配置,其 `relay_log_info_repository` 和其他变量的设置都不会影响恢复。
- 当基于传统模式(基于 file position)复制的情况下,请设置 `relay_log_recovery = 1` 和 `relay_log_info_repository=TABLE`。

2. 多线程模式的复制 (MySQL 5.7)。

- 当基于 GTID 模式复制的时候,并且设置了 `MASTER_AUTO_POSITION` 和 `relay_log_recovery = 0`,使用该配置,其 `relay_log_info_repository` 和其他变量的设置都不会影响恢复。
- 当基于传统模式复制的时候,请设置 `relay_log_recovery=1`、`sync_relay_log=1` 和 `relay_log_info_repository = TABLE`。

Semi_Sync Replication 方式的复制

基于传统复制,可能存在从库复制延迟的问题,那么当 MASTER 宕机后,SLAVE 可能处于落后的状态,如果这时 MASTER 无法恢复,只能使用 SLAVE 来替代 MASTER,就会导致数据丢失。在 MySQL 中,可以通过半同步复制来解决部分问题。

在半同步复制中,首先介绍几个重要的参数及它们是如何保证数据安全的。

参数

参数 `rpl_semi_sync_master_wait_point` 在 MySQL 的半同步复制中，控制 MASTER 在哪个阶段等待来自 SLAVE 的 ACK 确认。在 MySQL 5.7 中，该参数主要有两个值：AFTER_COMMIT 和 AFTER_SYNC（默认）。

- `rpl_semi_sync_master_wait_point = AFTER_COMMIT`：MASTER 将自己的 Binlog 写入到 Binlog 文件并且 sync，且向存储引擎提交事务，然后一直等待 ACK。当至少一个 SLAVE 接收到 Binlog 后，写入 relay-log 并返回 ACK 确认信息。MASTER 在接收到从库 ACK 确认信息之后，MASTER 将结果返回给客户端。
- `rpl_semi_sync_master_wait_point = AFTER_SYNC`（MySQL 5.7 新增）：MASTER 将自己的 Binlog 写入到 Binlog 文件并且 sync，之后会进入等待 ACK。当至少一个 SLAVE 接收到 Binlog 后，写入 relay-log 并返回 ACK 确认信息。MASTER 在接收到从库 ACK 确认消息之后，MASTER 向存储引擎提交事务，最终返回给客户端。

参数值对数据安全的影响

- AFTER_COMMIT：在该模式下，每次提交一个事务后（包括存储引擎的提交），等待从节点 ACK，所以 MASTER 上的其他 session 是可以看到这个提交的事务的。如果在等待 ACK 的过程中 MASTER 挂了，但此时 SLAVE 可能根本没有接收到相应的 Binlog，如果 MASTER 永远也启动不了了，那么实际上 MASTER 已经提交成功的事务在 SLAVE 上便会找不到了，也就意味着数据丢失了。
- AFTER_SYNC：在该模式下，事务在引擎层提交之前，等待 SLAVE 的 ACK 确认，如果这个时候 MASTER 宕机了，那么 SLAVE 可能已经接收到 Binlog 并且已经应用；如果 MASTER 永远也启动不了了，那么这个事务在 MASTER 上是不成功的，这时 SLAVE 就会多出这部分数据，可能导致问题。但是在 AFTER_SYNC 中，可以保证的是在 MASTER 上 COMMIT 成功的事务一定会同步到 SLAVE 上，确保数据不丢失。

MySQL 集群化如何保证数据库安全

MySQL 集群场景的集群化方案有两种：Galera Cluster 和 MySQL Group Replication。

Galera Cluster

Galera Cluster 主要用途是为 MySQL 提供一致性的集群化解决方案，它以插件的形式提供给 MySQL，并通过自身的 Write-Set 提供复制服务，从而实现 MySQL 的多线程并行复制和多源复制，且自带集群节点管理机制，可以主动监测集群节点状态，自动管理有问题的数据节

点, 同时还可以实现集群的多点写入和平滑扩容。

Galera Cluster 最关注的是数据的一致性。它对待事务的行为是, 要么在所有节点上执行, 要么都不执行。它的实现机制决定了它对待一致性的行为非常严格, 这也能非常完美地保证 MySQL 集群的数据一致性。

在集群中有一些特性需要注意, 比如所有的表都需要是 InnoDB 表, 如果不是 InnoDB 表, 将不会校验, 也不会传递, 会导致数据不一致的情况。在集群中, 只要不是所有的成员都发生宕机, 将不会出现数据丢失的情况, 因为数据的一致性有集群保证。那么, 如果集群中每一个数据库节点都发生宕机, 就需要参考上面所说的单机情况下是如何保证数据安全的。

上面说过, 在 Galera Cluster 中, 只要集群中所有的节点不同时发生宕机现象, 数据就不会丢失, 且是安全的。那么在集群中, 该如何配置一些参数, 来尽可能地保证性能最优。通过上面的介绍得知, 可以适当地调整如下三个参数设置, 使得性能更好一些。

- `innodb_doublewrite = ON`: 这个可以确保数据库在宕机后, 数据页是完整的, 不会出现数据库无法启动的现象, 这样节点在重新加入集群的时候, 可以避免 SST 操作, 只需要 IST 就可以加入到集群中, 缩短了故障恢复时间。
- `innodb_flush_log_at_trx_commit = 0`: 我们已经知道, 集群中的某一个节点 (只要不是全部同时挂掉) 宕机或其机器宕机后, 再次启动时, 都会从其宕机前的一致位置开始, 从其他正常节点处补数据, 那么不管这个节点是否丢数据, 只要是一致状态的数据库, 就可以将“丢失的数据”找回来。从这个意义上讲, 参数 `innodb_flush_log_at_trx_commit` 的值就无所谓了, 可以设置为最不安全的 0。
- `sync_binlog = 0`: 由于集群中的数据同步不依赖于落地的 Binlog 文件, 那么该参数可以调整为 0。

MySQL Group Replication

MySQL Group Replication 是 MySQL 官方提供的数据库一致性的集群化解决方案。MySQL Group Replication 是一个 MySQL 插件, 建立在现有的 MySQL 复制的基础上, 利用了 ROW 格式的 Binlog 和 GTID 等功能。MySQL Group Replication 基于 Paxos 协议实现, 在 Group 中, 内部允许部分节点挂掉, 只要保证绝大多数成员仍然存活且成员之间的通讯没有问题, 那么该 Group 对外就仍然提供服务。Group Replication 基于 Paxos 协议的一致性算法校验事务执行是否有冲突, 然后顺序执行事务, 达到最终的数据一致性。MySQL Group Replication 自带集群成员管理机制, 主动监测成员状态, 自动管理有问题的成员, 同时实现了集群多点写入与平衡扩容。

使用 MySQL Group Replication 时, 少数成员发生宕机同样不会造成数据丢失的情况, 除非所有的成员都发生宕机, 那么就需要参考单机如何保证数据安全。在集群中, 该如何设置一

些参数来保证性能和安全呢？通过上面的介绍得知，可以适当地调整如下四个参数设置，使得性能更好一些。

- `innodb_doublewrite = ON`：该参数还是需要开启，不然节点宕机后，可能导致数据库无法启动，这样修复起来会相对麻烦一些。需要使用备份，或者重新拉取数据。
- `binlog-format = ROW`：MGR 是基于行复制的，所以该参数必须设置为 ROW 格式。
- `master-info-repository = TABLE`、`relay-log-info-repository = TABLE`：在 MGR 中，复制的 `applier` 需要将 `master` 的信息和 `relay log` 的信息写入到 `mysql.slave_master_info` 和 `mysql.slave_relay_log_info` 表。这确保了 Group replication 插件具有一致的可恢复性和事务管理的复制元数据。
- `innodb_flush_log_at_trx_commit = 0`：我们已经知道，集群中的某一个节点（只要不是全部同时挂掉）宕机或其机器宕机后，在再次启动时，都会从其宕机前的一致位置开始，从其他正常节点补数据，那么不管这个节点是否丢数据，只要是一致状态的数据库，就可以将“丢失的数据”找回来。从这个意义上讲，参数 `innodb_flush_log_at_trx_commit` 的值就无所谓了，可以设置为最不安全的 0。

总结

数据库是用来存储数据的，把数据存入数据库中，就会要求数据是安全的、可靠的。所以，在数据库的发展过程中，安全性一直是数据库开发者重视的功能点之一。通过上面的分析也能发现，在 MySQL 中，针对安全性的设置非常复杂，在本章中只是列举了几个最常用的设置而已。

MySQL 的发展，可以说经历了三部曲，如下。

- 第一是 MySQL 单实例的情况，因为 MySQL 一开始的设计并不是分布式或集群化的，这也是所有传统关系数据库的共性。所以，开发者会在数据库自身功能上花费巨大的精力去解决数据安全性的问题，他们为使用者设置了各种各样的可配置参数，这就有了两次写、重做日志等这样的机制。
- 第二是 Replication 时代。这是 MySQL 走上集群化的重要一步，它为 MySQL 集群化提供了一个非常简单的实现方式。这一步虽然迈出得很早，但迈得并不彻底，而且有些沉重。仔细审阅 MySQL 的发布日志就会发现，Replication 的各种 Bug 层出不穷，而且 Replication 也不断地修改、更新。为了保证 Replication 功能的安全实现，MySQL 也给出了很多很多参数，可以通过这些参数实现自己的需求。当然，这是比较复杂的。
- 第三是真正的 Cluster 时代。也就是随着 Galera Cluster 及 Group Replication 的到来，真正迎来了 MySQL 的集群化时代。通过上面的介绍会发现，在集群化的配置中，单机或

单实例的安全性其实已经没有那么重要了。由于数据是分布在集群中多个实例上的，是多份存储，并且集群是实时同步的，只要整个集群不是全部同时宕机，基本就不会存在数据安全问题。

所以，MySQL 在线上环境中，建议尽量保证集群化，这样可以最大限度地避免单机故障时线上数据不可用，并在集群范围中保证数据不丢失。最后，请做好备份，并且保证备份是安全、可用的，这一点尤为重要。

MySQL 性能拾遗

前面谈到了 MySQL 的数据安全问题，解决了 MySQL 作为数据库怎么能安全地存储数据这个大问题。接下来，来讨论另一个重要的问题，那就是 MySQL 的性能问题。其实，安全和性能在很多情况下是相互矛盾的。追求安全，一般情况下都会以牺牲性能为代价；反之，追求性能的时候，安全也很可能会受到影响。当然这不是绝对的，还有很多相互独立的影响因素，需要深入了解，才能用好 MySQL。

这里先明确一下，什么才是我们关注的性能问题。我们所说的 MySQL 性能是指用 MySQL 存储和查询数据的效率。这可以用 TPC-C 等测试模型进行测试，用 TPS 或 QPS 来量化。作为 MySQL 的使用者，我们当然是期望 MySQL 提供尽可能大的 TPS 或 QPS，通俗地说，就是期望 MySQL 的性能尽量好，执行 SQL 语句尽量快。

影响 MySQL 性能的因素很多，拟从以下几个方面谈起。

适当的数据文件大小

不管是什么数据库，数据量大了之后，都会影响数据库的性能，这是很容易理解的。毕竟，操作数据最终会归结到物理的扫描，数据量越来越大，数据文件的大小也会不断增长，索引文件也越来越大，在对 B+ 树搜索时，可能会因为其层数不同，对性能有所影响。更为严重的是，如果 SQL 语句没有用到索引或索引不够优化，需要扫描数据文件的时候，性能就会随着文件大小的增长而急剧下降。

另外,数据文件大到一定程度之后,在 DBA 进行 DDL,或者备份、迁移等日常运维工作的时候,会有很大麻烦,不仅延长了运维时间,还有可能造成不可预期的风险。所以,在很多数据库的使用规范里面都有一条,即限制数据库单表的数据量和数据文件大小。

碎片空洞问题

关于数据库数据量大了之后怎么分库分表的问题,不在此讨论,请参考第 24 章。这里要重点说的是另一个容易忽略的问题,即数据文件空洞的问题,主要关注 InnoDB 引擎。InnoDB 在执行数据的修改操作,例如删除一行数据时,表面上看到的是数据库返回删除成功,但在底层并非如此。实际上,这行数据在物理上只是被“标记”为删除,并不从索引和数据文件中真实删除,所以,它所占据的空间也没有真正释放。InnoDB 后台有个 Purge Thread,会定期去清理这些已经删除的索引和文件。即便如此,InnoDB 也不会回收这些已经删除的空间给操作系统,这就会在数据文件中存在很多空洞,这些空洞很可能会一直存在,并且越来越大。

曾经碰到过一个案例,有个表叫 `news_deliver`。它的作用是用来暂存需要分发的消息,等消息过时之后,就会被自动清空。所以,这个表会有大量的插入和删除操作,但它实际的数据量不会太大。但经过了一段时间之后,发现这个表的数据文件越来越大,访问的效率也急剧下降。通过 `SHOW TABLE STATUS` 发现这个表的碎片或空洞非常严重,判断的办法是通过 `SHOW TABLE STATUS` 的结果,查看实际数据 `Data_length` 和空洞数据 `Data_free` 的比例。如果 `Data_free` 非常大,则说明这个表需要优化了,优化的方法是:用 `OPTIMIZE TABLE <table>` 或 `ALTER TABLE <table> ENGINE=InnoDB` 来重建表空间。

为了及时发现并修复碎片空洞问题,在内部的巡检平台上特意加上了碎片巡检这一项,其原理就是根据上面提到的 `Data_length` 和 `Data_free` 的数值来判断的,强烈建议 MySQL 的使用者在平时的生产过程中都要关注这个问题。

设计问题

除了上面所说的,要注意碎片空洞问题外,在设计和维护数据库表的时候,平时注意一些细微问题,也有助于尽可能地控制数据库数据文件的大小。

1. 字段的合理设置。

在设计表的时候,要合理地选择正确的字段类型,根据需要而定,不能过度滥用,有个原则是:尽可能使用最小的数据类型,最小的也是最有效的。例如,所有的数字都用 `Bigint`,所有的字符串都是 `varchar (1024)` 或直接用 `Text` 等,这都是不对的。仅仅是数字类型,`MEDIUMINT` 就比 `INT` 节省四分之一的空间。有个著名的特例,就是 IP 类型的存储,惯例上会用字符串存储点分十进制数据,而 IP 本质上却是一个无符号的整型数字,这就只会用到原来需要空间的四分之一了。

还有需要注意的是 NULL 的默认。有个规则是，在设计表的时候，尽可能地设置字段为 NOT NULL。这样在检索数据的时候，不需要再额外逐个值去判断是否为 NULL，从而会更好利用索引，加速 SQL 的执行速度。同时，每个字段会节省 1bit 空间。

2. 行存储格式。

在 MySQL 5.7.9 之后，InnoDB 表的行格式由之前默认的 COMPACT 变成了 DYNAMIC，关于此二者及其他格式的区别，可以查看 MySQL 在线文档。这里要说的是，总体上，COMPACT 格式会节约 20% 的空间，同时，在存储 UTF8 或 UTF8MB4 数据的时候，COMPACT 格式在存储时会尽量地节约空间，不对其中的空格进行存储。这与其他存储固定长度数据的类型相比，也会节约不少空间。所以，在一些关键字段的格式选择上，如果有空间问题，可以特殊对待，而不要一味地使用默认值。当然，InnoDB 还有一种存储格式为 COMPRESSED，相比 MyISAM，InnoDB 的压缩格式可以同时支持读和写，使用上更方便一些。

3. 索引问题。

下面还会详细谈到 MySQL 索引的优化，这里要说的是，添加索引的时候一定要注意它带来的副作用。索引虽然能提升查询的性能，但由于需要额外写索引文件，它会降低 MySQL 写数据的速度，同时，也会增大数据文件的大小。加的索引越多，写入数据越慢，数据文件越大。所以在设计索引的时候，会要求主键的字段类型一定要用数字类型，并且要尽可能地小，能用 INT 的，绝对不要用 BIGINT。在设计二级索引的时候，只添加最需要的索引，避免添加重复索引和冗余索引，针对长字符串字段，尽量添加前缀索引。

合理设计表结构

在设计关系数据库表结构的时候，教科书上有著名三范式的说法。简而言之，第一范式要求表不能有重复的列；第二范式要求表有主键，所有非主键字段都需要依赖主键；第三范式要求表中的字段不能依赖于同一表中的其他非主键字段。

在实际生产中，使用数据库的目的是最大程度地适应业务的需求和发展。有时候，需要根据实际情况合理设计表结构，就不能严格地按照范式来设计，这就会出现一些反范式的情形。

冗余存储

在做社交网站的时候，有一类表非常重要，叫好友关系表。基本上，两个用户 ID 就可以确定一个好友关系，可以定义表结构为：(user_id, friend_user_id)，用来存储一对好友关系。

但是，在实际的业务中，可能会有以下需求。

user_id	friend_user_id	other_info
1001	1002	other_info
1001	1003	other_info
1001	1004	other_info
1004	1007	other_info
1004	1008	other_info
1005	1003	other_info

1. 查询某人的好友。

从上面的表中可以看出, 如果需要查询某个人所有好友的话, 需要在 `user_id` 和 `friend_user_id` 这两个字段里各做一次查询, 然后把数据合并才行, 例如: `where userid=1004 or friend_user_id=1004`。这样的 SQL 显然不能很好地利用索引。另外, 如果觉得这个表比较大, 需要水平切分的话, 那么这样的需求就很难实现了。这种情况下就不得不冗余存储, 把好友关系按照每个人的方向都存储一份, 如下表所示。这样在查询某人的好友列表时只需要 `where user_id=1004`, 就可以很高效地返回所有需要的记录了。

user_id	friend_user_id	other_info
1001	1002	other_info
1001	1003	other_info
1001	1004	other_info
1002	1001	other_info
1003	1001	other_info
1003	1005	other_info
1004	1001	other_info
1004	1007	other_info
1004	1008	other_info
1005	1003	other_info
1007	1004	other_info
1008	1004	other_info

类似的业务还有电子商务中的订单信息, 消费者需要查询自己的所有订单, 商户也需要查看同样的订单。本来可以在一条记录中存储所有信息, 但是因为业务需要, 而且数据量非常巨大, 所以只好分开双向存储。

2. 查询特别好友。

有个业务允许用户在自己的所有好友中设置少量几个特别好友，对他们的信息，可以特别关注。这其实可以在上面的好友表里，通过一个字段去定义此好友是否为特殊好友，需要查询自己特殊好友的时候加个布尔型的判断就可以了。但是在实际业务中，好友关系表有几百亿之巨，对付这样一个查询，真是不轻松。后来为了满足业务需要，要把这类信息单独列出来，冗余存储一个 `special_friend` 列，只在这一个表里存储特殊好友，并且去掉了布尔型判断这样的低效率索引，查询效率提升立竿见影。

类似的，还有个人主页的最新留言。在多 UGC 的网站中，个人主页既有图片，又有博客，还有视频等各种内容，每种内容都可以供人评论。本来在数据库设计上，这是一个垂直拆分的典型案例，不管有多少种 UGC，分类存储就行了。然后把每种内容的评论也分类存在不同的表里面。但是没准有一天，有个产品经理说，需要给用户提供一个综合表，冗余存储最新信息，需要查询的时候，直接从此表中取就可以了。

拆分存储

最典型的为了性能而进行拆分存储的例子是，当表中有 Blob 或 Text 字段，表的更新和访问特别频繁，且这样的字段可能在大多数情况下都不会被选取时，就可以把这样的字段拆出来单独存储，通过相关的键值关联即可。

这里举另外一个例子。有一张表需要存储用户的生日，那么通常的设计应该是如下这样的。

user_id	birth_date
1001	'1989-05-21'
1002	'1981-12-13'
1003	'1979-04-21'
1004	'1981-12-13'
1005	'1986-05-21'
1006	'1987-11-13'

这是一个标准的做法，需要查询某人生日的时候，通过 `user_id` 直接匹配就可以。但是，问题来了，业务要做得有声有色，就得提供各种人性化的功能，例如以下几点。

- 给用户推荐同年同月同日生的其他“有缘”用户。这个貌似还好办，通过 `birth_date` 查询不同的 `user_id` 就可以解决。
- 给用户推荐同月同日生但是不同年的“有点有缘”用户。这就麻烦喽！看来得去查查 MySQL 的日期计算函数了，另外，等一等，如果 `birth_date` 在计算函数中的话，就不能

用到索引了，这比较尴尬！

- 给用户推荐相同星座的“有缘”用户。这也不太轻松啊！

为了满足这个需求，只好把表设计成如下这个样子。

user_id	birth_year	birth_month	birth_day
1001	1989	05	21
1002	1981	12	13
1003	1979	04	21
1004	1981	12	13
1005	1986	05	21
1006	1987	11	13

上面的结构，一目了然，我们可以很轻松地说：“还有什么需求，放马过来吧。”

重复存储

在对数据库进行拆分之后，我们会发现新的问题。例如有一些存储基础配置的表，本身数据量不大，但非常重要，并且在进行某些查询的时候，需要被别的业务表 JOIN。但一般业务的表会越来越大，很可能就被拆分成好多份，分布在多个实体机器上，这时由于受限于 MySQL 不能跨实例关联，即便是最高效的 JOIN 也没办法做了。这该怎么办呢？解决思路有两个，一是不再进行关联，把这个逻辑放在业务端来做。但这个时候，业务程序需要修改，这无疑给开发人员增加了负担，给业务发展拖了后腿。另一个思路，就是重复存储，这不同于上面说的冗余存储。这是把某个表或某一部分表，重复地存储在多个 MySQL 实例上，以便查询的时候进行本地 JOIN 操作。

特别提醒

以上说到的几个特例，都是为了满足业务需求，在不符合数据库范式的情形下采取的求全之策。事实上，MySQL 也支持外键约束，但在实际业务中，特别是互联网业务需要高并发、高性能的场景中，一般都不会使用外键约束，这对数据一致性问题提出了挑战。同样地，以上所述的各种特例，也存在数据一致性的问题。一般的做法是，在业务层会有专门的逻辑或解决方案来保证数据的一致，以最终一致的时差来换取即时访问的性能问题。这种做法是有局限性的，需要我们时刻了解自己的业务需求，并作出准确的判断——哪些场景可以这么实现，而哪些场景要求数据必须时刻一致，不能这么处理。


正确使用索引

索引对于 MySQL 来说至关重要, MySQL 之所以能够高效地检索数据, 可以说全赖索引之功。关于索引的原理, 在前面已经详细地介绍过了, 此处不再赘述。这里要强调的是, 正确地使用索引, 才能让 MySQL 发挥其最大的能力, 实现高性能查询。MySQL 的性能调优, 永远做不完的事情就是发现慢查询, 为新增 SQL 添加合理索引。当然, 业务不再更新除外。

想要正确地使用索引, 就首先要理解 MySQL 的索引原理。前面有专门的一节来介绍 InnoDB 的索引原理实现, 这里不再多说, 只强调如下几点。

1. MySQL 在使用索引时, 采用的是最左匹配原则。

这如果是单列索引则很容易理解。如果是多列索引, 例如 `idx_a_b_c(a,b,c)`, 则可以发挥索引功能组合为 `(a)`, `(a,b)`, `(a,b,c)`, 并且索引是一次遍历没有回溯的, 所以如果要用到两列或两列以上, 那么除了最后一列外, 前面的都需要精确匹配才行。因此, 下面的 SQL 可以用到索引。

```
 select * from t order by a,b,c;


select * from t where a=CONSTANT order by b ASC,c ASC;

select * from t where a=CONSTANT order by b DESC,c DESC;

select * from t where a=CONSTANT and b=CONSTANT order by c;

select * from t where a=CONSTANT and b>CONSTANT order by b;
```

下面的情况用不到索引或用不到索引进行排序。

```
 select * from t order by a,d;


select * from t where b=CONSTANT order by a;

select * from t where a=CONSTANT order by b ASC,c DESC;
```

需要提示的是, MySQL 8.0 已经支持 Order By 混合排序了, 所以 MySQL 的索引在 8.0 之后会支持 Order By `a desc, b, asc, c desc` 这样的排序语句。

2. MySQL 在计算列里无法使用索引。

例如下面的 SQL 不能使用索引。

```
 select * from t where ABS(a)=CONSTANT;

select * from t where f(a)=CONSTANT; //f为任意函数
```

3. MySQL 在否定条件中不能使用索引。

例如, where 条件里面有 <>、not in、not exists 的时候, 即便是在这些判断字段上加上索引, 也不会起作用。


4. MySQL 在 join 中连接字段类型如果不一致, 则不能使用索引。

这很容易理解, INT 类型的字段没办法和 VARCHAR 类型的字段 JOIN。有个例外是 CHAR 和 VARCHAR 如果在定义表的时候长度一样, 就可以利用索引 JOIN, 反之不行。例如, CHAR(20) 和 VARCHAR(20) 可以利用索引, CHAR(20) 和 VARCHAR(25) 则不行, 不管 VARCHAR 里面实际存储的值是多长。

另外, 如果两个字段列的字符集不同, 则不能 JOIN。我们曾经在 UTF8 和 UTF8MB4 转换的过程中碰到过这样的问题, 本来工作良好的索引, 把其中一张表的字段单独修改为 UTF8MB4 后, 索引失效了, 引起了慢查询故障。

5. 机智地利用覆盖索引。

我们知道, 在 MySQL 利用 B+ 树索引检索数据的时候, 如果不是基于聚簇索引, 或者说如果不是基于主键的检索, 那么即便是 SQL 语句能够利用索引, 但索引返回的信息也只是所需结果行的主键值, 要取得全部数据, 还需要通过这些主键值重新到数据文件里再做一次检索操作, 这样就需要额外的 IO, 降低了查询效率。如果能优化 SQL 或索引, 让 MySQL 只需要通过索引就可以返回所需的数据, 而不必回表, 从而提升效率, 这就是覆盖索引, 或者叫 Covering Index。如果 Explain 的 Extra 列的值为 Using Index, 则表示目前 MySQL 正在使用覆盖索引。由于覆盖索引减少了一次回表操作, 所以它带来的性能提升是显而易见的, 特别是对高频查询的 SQL, 有时通过修改索引可以起到事半功倍的效果。举一个简单的例子, 如下。

```
 select namespace from dbrequest_info where update_date between '2017-01-01' and '2017-02-01';
```

在 Inception 中, 有个查询一段时间更新业务状况的查询, SQL 如上所示。如果在 dbrequest_info 表里有索引 (update_time), 则可以利用索引进行查询。如果想要利用覆盖索引的话, 可以添加索引 (update_time, namespace), 则在 Explain 的 Extra 里面会显示 Using Index。

这里要提醒的是, 虽然覆盖索引提升了查询性能, 但是由于增加了索引字段, 会对写入数据造成一定的影响, 同时会占用更多的空间, 所以在决定是否使用覆盖索引的时候, 一定要权衡利弊。建议在一些高频的查询上使用覆盖索引, 而不是滥用。

6. 其他一些需要注意的情况。

如果某个索引的分辨率不高, 即便是在利用索引的情况下, 扫描全表内容也经常超过 20%, 那就要谨慎添加索引, 例如: 性别、星座、等级这类信息。在 MySQL 中尽量避免使用 like 查询, 特别是 like '%name', 这种左边模糊匹配的情形, 使得 MySQL 无法利用索引。如果出现隐式的字符类型转换, MySQL 也不能利用索引。这就相当于在判断列上加了函数一样。

想要正确地使用索引,就要看明白 MySQL 的 Explain。有关 Explain 的详细介绍可以参考官方手册,这里需要重点关注的是 Extra 列里的 Using filesort 和 Using temporary。如果在 Explain 的结果里出现这二者或二者之一,那么很不幸,这是 MySQL 里面影响查询性能的大杀器,需要重点关注。

MySQL 系统参数

MySQL 参数繁多,根据自己的需要和实际场景可以自由调整,这里重点讨论一些比较常用的参数。

- **general_log**: 建议在数据库正常服务时,将这个参数设置为关闭状态,因为它会记录提交到 MySQL 的一切东西,既浪费磁盘又影响效率。同时它又是分析问题的利器,如果觉得数据库有莫名的访问或 SQL,可以把它打开,截取一段时间的日志,以帮助定位问题。
- **query_cache_size**: Query Cache 是用来缓存 SQL 语句文本和对应查询结果的缓存空间。如果相应的表没有变化,那么下次再碰到完全一致的 SQL 的时候,则跳过一切解析和查询,直接返回结果。这对某些情况是适合的,但是如果表变化非常频繁,SQL 也是动态生成的,则由于要不断更新 Cache 中的内容,并且这个时候锁粒度非常大,反而会成为瓶颈。所以很多情况下,会关掉这个选项,方法是设置此参数为 0。
- **sort_buffer_size**: MySQL 在排序的时候,如果用不到索引就会在内部临时进行排序,这时候会用到排序的 buffer。如果这个参数值过小的话,排序过程中会把结果写入到物理磁盘,严重影响效率。但是,这是一个 session 级变量,也就是说每个 session 在用到它的时候,都会申请一块这么大的内存,所以如果太大的话,又很可能会耗尽物理内存,导致服务器 OOM。在设置它的大小时,一定要根据自己的实际情况,灵活设置。类似的另一个参数是 join_buffer_size,是在 join 无法用到索引的时候用到的 buffer,也需要灵活设置。
- **tmp_table_size**: 在 Group By 或 Distinct 的时候,如果 SQL 语句用不到索引,就会使用系统内部临时表记录中间状态。如果 tmp_table_size 不够大,则 MySQL 会自动使用物理磁盘,这会对查询性能造成很大的影响。增加此参数可以降低这种情况发生的概率。



注意: 这是占用物理内存的,需要考虑实际的内存空余情况。

- **innodb_buffer_pool_size**: InnoDB 最重要的缓存,用来缓存 InnoDB 索引页面、Undo 页面及其他一些辅助数据,在第 11 章已经详细介绍过它的原理。它的大小是影响性能的重要因素,基本上各种文档都会要求在内存允许的情况下尽可能地配置大一些,官方则建议配置为物理内存的 50%~75%。

- `innodb_buffer_pool_instances`: 通过这个参数, 把原来一整块 Buffer Pool 分割为多块内存空间, 每个空间独立管理自己的空闲链表、刷新链表、LRU 及其他数据结构。这大大增加了并发性, 能更有效地利用缓存。
- `innodb_log_file_size` 和 `innodb_log_files_in_group`: 这两个参数结合, 决定了 REDO 空间的大小。REDO 空间越大, 可以存储的增量更新日志越大, 有效降低 Buffer Pool 脏页面被淘汰的速度, 同时减少了 checkpoint 的次数, 降低磁盘 IO 置换率, 从而提升数据库的写入效率。不过也有可能会导致数据库异常退出时, 恢复时间被拉长。
- `innodb_old_blocks_pct` 和 `innodb_old_blocks_time`: 这两个参数控制 Buffer Pool 中缓存数据的过期和移动行为, 二者结合设置, 可以优化一些全表扫描带来的大规模更新 Buffer 等问题。
- `innodb_numa_interleave`: MySQL 服务器很多时候会出现内存被交换到 SWAP 分区的情况, 这时性能就会急剧下降。但是当我们去查看服务情况的时候, 往往会发现实际上操作系统还有很多空闲内存。在 MySQL 5.7.9 之后, 通过这个参数可以避免这个问题。MySQL 在分配内存的时候, 会把 NUMA 的策略设置为 `MPOL_INTERLEAVE`。当然, 这需要在支持 NUMA 的系统上编译 MySQL 才行。
- `innodb_autoinc_lock_mode`: 在 InnoDB 有自增列的情况下, 在插入数据的时候, 会自动产生自增值, 这个参数是控制自增值生成的方式。目前有三个选项: 0、1、2。其实这是个枚举, 真正的含义是: “traditional”、“consecutive” 和 “interleaved”。使用 2 即 “interleaved”, 这样在 INSERT 数据的时候, 不会用到表级的 AUTO-INC 锁, 避免 AUTO-INC 的死锁问题, 在 INSERT ... SELECT 的场景下会极大地提升性能。在做普通 INSERT 的时候, 也会提升并发执行的效率。



注意: 这个时候产生的自增值不是连续的, 同时 Binlog 格式需要设置为 ROW, 才能保证数据的安全和一致性。

- `innodb_flush_method`: InnoDB 刷数据和日志到磁盘文件的方式, 默认为 NULL, 但其实如果是在类 UNIX 系统上, 默认为 `fsync`, 在 Windows 系统上默认为 `async_unbuffered`。这里要说的是, 它还有个可能的值是 `O_DIRECT`, 在使用 SSD 或 PCIE 类型的存储时, 可以设置为 `O_DIRECT`, 底层调用 `directio()`, 直接修改写入磁盘, 以提升性能。
- `innodb_doublewrite`: 关于 InnoDB 的两次写, 之前也详细介绍过其原理。这里要强调的是如果底层存储是支持原子写的, 则可以关闭两次写, 以提升效率。这跟上面的 `innodb_flush_method` 是一样的。
- `innodb_io_capacity`: 之前说过, InnoDB 有后台线程在不断做 Flush 操作, 影响这个操作频率的就是 `innodb_io_capacity` 这个参数, 如果碰到系统因为后台 Flush 操作而产生周期性性能降低的情况, 特别是在使用 SSD 设备的时候, 可以适当提高这个参数的值, 以加

速 Flush 的频率。

- `innodb_thread_concurrency`: 在并发量大的实例上, 增加这个值, 可以降低 InnoDB 在并发线程之间切换的花销, 以增加系统的并发吞吐量。
- `innodb_flush_log_at_trx_commit`: InnoDB 刷日志文件的方式, 之前已经详细介绍过了。在 0、1、2 的选型中, 0 性能最好, 但最不安全, 不推荐。1 最安全, 但性能最差, 如果使用的磁盘足够好, 可以弥补性能损失的话, 还是建议为 1, 否则, 可以根据实际情况选择性地设置为 2。
- `sync_binlog`: MySQL 同步 Binlog 到磁盘的方式, 之前也有过详细介绍。1 最安全, 但性能最差。0 性能最好, 但最不安全。如果能在物理磁盘或架构上做出弥补, 还是建议设置为 1。
- `binlog_format`: MySQL 日志格式, 从 MySQL 5.7.7 之后, 官方就默认为 ROW 了。这也是在很多场景下都建议的选项。
- `binlog_order_commits`: 事务在提交的时候写入 Binlog 的顺序。这是把双刃剑, 如果打开, 可以保证事务都以相同的顺序写入二进制文件, 如果关闭则可以提升性能。需要根据实际情况决定。
- `tx_isolation`: 设置 MySQL 隔离级别, READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ、SERIALIZABLE 这四种级别越来越严格, 但性能也越来越差。在 MySQL 中推荐与 `binlog_format` 结合设置, 推荐隔离级别设置为 READ COMMITTED, 这在保证性能的前提下, 同时设置 `binlog_format=ROW`, 确保通过 Binlog 同步数据主从库的一致性, 兼顾到安全, 满足绝大多数业务的需求。
- `slave_parallel_workers`: 在进行多线程复制的时候, 如果设置此参数为非零值, 则可以打开多线程并发执行回放日志的操作, 以提升 Slave 的同步性能。

内存和 CPU

操作系统配置的差别, 对 MySQL 性能的影响是相当明显的。关于操作系统的优化, 可以通过互联网或其他书籍找到很多参考资料。在性能上, 需要提醒一下关于 CPU 和内存的使用。关于磁盘的部分, 下面会单独讲。

1. 关于 CPU。

MySQL 在执行单个 SQL 语句的时候, 在底层只能用到一个 CPU, 所以要想提升 SQL 的执行效率, 在考虑 CPU 的时候, 最好选择主频高的, 这有利于加速单条 SQL 语句的执行效率。在有些极端条件下, 例如电子商务中的秒杀情形、关键节日集中庆祝的时刻及一些访问密集型业务, 并发需求很大, 这个时候, 并发执行的 SQL 需求很大, 要保证在短时间内有巨大的吞吐量, 就需要选择核心数多的 CPU。

2. 关于内存。

内存是解决磁盘 IO 能力不足,有效提升性能的利器。在大部分情况下,增加 MySQL 主机内存,并配置到数据库引擎中去,可以有效提升性能。关于 MySQL 内存的配置项,上面已经提到。但也有例外场景,我们曾经遇到过这样的场景,数据库所配置的物理内存已经远远超出了数据库实际的数据大小,性能仍然不能满足需要,即便是再去增加内存,也无济于事,这时就需要从 SQL 语句优化、表设计的合理性等软件因素去寻求解决途径。

还有一个特例,是我们曾经处理有关 MySQL Cluster 性能问题的场景。我们知道,MySQL Cluster 的存储引擎是 NDB,它是基于内存存储的。有很多案例是,项目在开始的时候选用了 MySQL Cluster 的方案,并且实施之后运转良好。但是,随着时间的推移和业务的发展,数据量越来越大,MySQL 的性能下降很快。在这种情况下优化性能的时候,就不能考虑增加内存的情况,因为这时的瓶颈多数情况下不是 IO 的问题,如果盲目增加集群机器,扩大整个集群的规模和内存总量,并不能解决问题,反而性能可能会更差。其原理是,NDB 是自动切分做分布式存储的,并且都是把数据存在内存中,其数据的存取效率很难再提升,性能影响主要来自 SQL 语句中查询逻辑的处理和数据的集中合并,如果盲目增加集群机器,可能会造成重新分布分片存储的情况,并且分布更加稀疏,合并数据的时候需要更多的资源,从而性能也就更差了。

值得注意的是,在选择内存的时候,要注意内存频率跟主机 CPU 频率的匹配,二者一致才能发挥其最大处理能力。

磁盘的革命

由于数据库是数据存储系统,数据最终落地的载体是磁盘,或者说硬盘,所以硬盘所能提供的 IO 能力也直接影响了数据库的性能。在 MySQL 里面,大量的选项、配置和算法都是为了优化磁盘 IO 而做的,大致的原则就是把存取频繁的数据在面对客户的时候由磁盘存储变成内存缓冲,而由于机械磁盘顺序旋转访问的特点把对磁盘操作的分散随机访问转变为集中顺序访问。传统机械磁盘的原理请参考相关文献,我们要强调的是,在选择此类磁盘时,要注意磁盘的转速,转速越大,它能提供的 IO 能力也越强,性能可能也会更好。在配置磁盘 RAID 的时候,建议采用 RAID10 配置,尽管 RAID10 会更浪费磁盘空间,但它提供了更好的性能。直观地讲,采用 RAID5 时,由于需要更新校验数据信息,所以每写一次数据,都需要读取数据及奇偶信息,经过计算后,再更新校验数据,然后再写入实际数据,真是百转千回。而 RAID10 则直接写入数据。

值得庆幸的是,随着磁盘科技的发展,很多企业已经选择用新兴的基于 Flash 的 SSD 存储作为数据库存储硬盘了。在优化硬件对数据库性能的提升上,从来没有像闪存这样对数据库影响有如此之大的。过去工作那么多年,由于它的存在,帮我们解决了很多问题,这里有必

要着重介绍一下。

SSD 技术的基本概念

SSD (Solid State Disk 或 Solid State Drive, 简称 SSD, 俗称固态硬盘) 是一种基于永久性存储器 (如闪存) 的计算机外部存储设备。固态硬盘已经逐渐在计算机和存储系统中代替传统机械硬盘, 虽然在固态硬盘中已经没有可以旋转的盘状结构, 但是依照人们的命名习惯, 这类存储器仍然被称为 “硬盘”。

SSD 的物理接口上有传统的 SATA 接口, 在更高性能要求的场景中, 企业会更倾向于选择拥有更高带宽和更低延迟的主板 PCIe 接口, 以及 PCIe 另一种物理形态的可热插拔的 U.2 (SFF-8639) 接口。

图 26.1 是一款 PCIe 接口的 SSD 产品, 它看起来像传统的网卡或显卡, 但图中密集的黑色小块部分暴露了它的身份, 那是 Flash 存储单元, 是 SSD 存储数据的载体, 目前最大单盘可以做到 12.8T, 这么大的容量可以降低数据库因为空间问题拆分的几率, 也有效地提升了单个服务器其他硬件资源的使用率, 大大节约了成本。当然, 这仅仅是 SSD 硬盘的优势之一, 更多的细节在下面会讲。

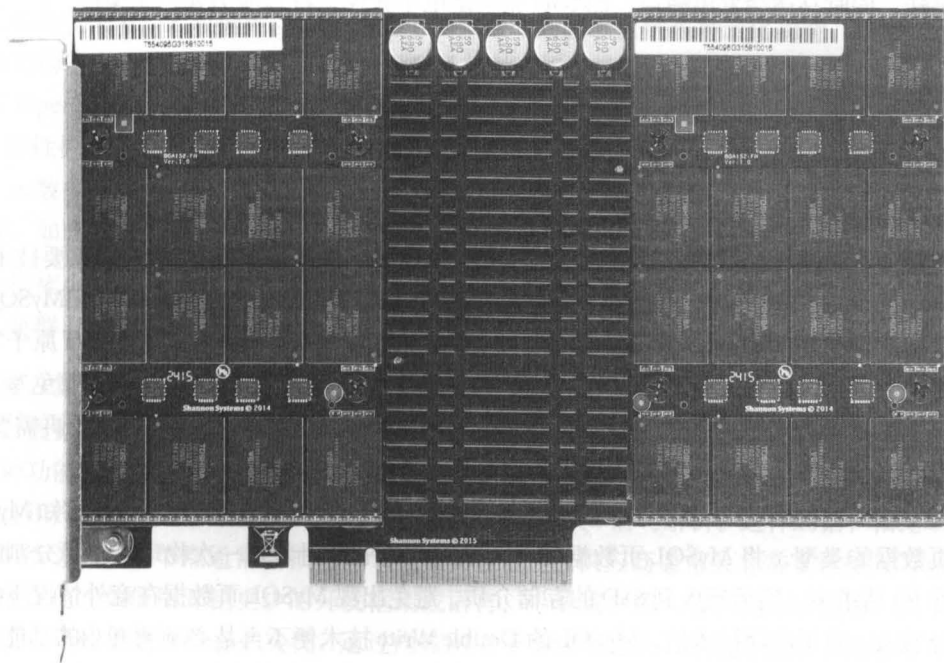


图 26.1

SSD 相比传统磁盘的优越性及数据对比

首先看一下 SSD 的一些诱人的数据和特点。

- 高 IOPS (Input/Output per Second)，即一秒钟能够完成的读写次数，此项指标是存储性能最直接的表现。传统磁盘每秒钟差不多可以完成 200 次 IO 请求，而 SSD 每秒钟可以完成高达 60 万次。
- 低延迟，即主机下发存储指令后，存储介质完成存储并返回正确应答的时间，传统磁盘由于需要驱动马达转动盘片和摇动磁头手臂，完成单个 IO 需要 2000 微秒左右，而 SSD 只需要不到 100 微秒。
- 低功耗，使用磁盘创建接近 SSD 性能的磁盘阵列总功耗高达几百到上千瓦；而一块 PCIe SSD 的功耗不会超过 25 瓦。
- 大容量，传统企业级磁盘容量在 4TB 以下，并且容量增长困难，而成熟的企业级 PCIe SSD 已经有单盘 12.8TB 的容量，并且会按照摩尔定律继续增长。
- 故障率低且故障模式可预测，传统磁盘由于是机械部件，故障不可预测，年故障率在百分之三左右，而 SSD 的故障可以预测，年故障率在千分之五以下。
- 抗震能力强，且无噪声，传统磁盘是机械旋转设备，对震动敏感，震动甚至会引发磁盘停转，同时马达会发出噪声，而 SSD 完全是电子元件，抗震能力强，无噪声。

SSD 在 MySQL 上的优化

SSD 的出现给 MySQL 的优化带来了新的思路，下面从其技术特点谈起。

1. 原子写技术。

在前面花了很大篇幅来介绍两次写 (Double Write)，这也是 MySQL 的一个重要技术点。MySQL 使用 Double Write 技术来避免因存储意外离线而造成的数据丢失。当 MySQL 的数据存储使用 SSD 时，情况就完全不同了。由于很多 SSD 产品现在都支持了原子写技术，也就是说，它在物理上就可以支持一次写入的原子性，这可以从本质上避免写入过程中的页断裂或数据丢失问题。也就是说，两次写这个特性在 SSD 上已经不再需要了，但是这个特性会造成一定的性能损失和 SSD 寿命的缩短。

基于主机构架的 PCIe SSD (如图 26.1 引用的宝存科技 Direct-IO 系列) 可以感知 MySQL 页数据的类型，将 MySQL 页数据的写入操作，原子地封装到一次物理上不可分割的硬件 IO 请求中，然后写入到 SSD 的存储介质，避免出现 MySQL 页数据在意外情况下的部分写入。应用此项技术后，MySQL 的 Double Write 技术便不再是必须要开启的功能，此举可以大幅度提升 MySQL 数据写入的延迟，同时延长一倍的存储介质使用寿命。

2. SSD 对 Galera Cluster 的影响。

在本书的第二部分介绍了基于 Galera Cluster 的 MySQL 高可用方案。在这个方案里，除了利用了 Galera Cluster 自身的优点，解决了 MySQL 集群间的数据一致性之外，也利用 SSD 高效率的特性，规避了 Galera Cluster 由于需要一致性而带来的性能损失的问题。由于 Galera 集群中不同节点之间需要保持状态一致性，任何一个节点都不能“落下”太多，否则就会带来 Flow Control 问题。提升所有节点的效率，特别是提升写入硬盘的效率，可以有效缓解这种情况，所以我们在集群中限定性地采用了 PXC+SSD 的方案。

云中漫步

云计算是一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问，进入可配置的计算资源共享池（资源包括网络、服务器、存储、应用软件、服务），这些资源能够被快速提供，只需投入很少的管理工作，或服务供应商进行很少的交互。云计算的服务能力在理论上可以认为是按需提供、取之不尽、用之不完。所以我们在谈及性能的时候，把云计算请出来，相信云服务可以提供更好的数据库服务。

云上 MySQL，准确地说应该是 DBaaS，它是基于 MySQL 提供的一种数据库服务，如腾讯云的 CDB、AWS 的 RDS、阿里云的 RDS 等。它有另外一个更贴切的洋气名字：云数据库。什么是云数据库呢？从产品的角度来讲，云数据库应该是满足云计算特征的数据库服务；从技术的角度来看，云数据库应该是深度定制数据库内核来提供各种云化功能。通常，云数据库的架构包含实现云化功能的管控平台和承担处理 SQL 请求的云数据库内核。管控平台有基于 Openstack Trove 来构建的，也有自行设计的。管控平台主要解决数据库管理、备份回档、弹性扩展、线上调优和安全审计等问题，它是面向用户的云数据库控制台的底层支撑平台。云数据库的内核，不仅仅是 MySQL，可能还会包含接入集群、实例集群和数据库存储集群，如图 26.2 所示。

云数据库内核技术栈基于云数据库的架构，云数据库的性能优化会覆盖到多个子系统，如接入集群、实例集群和数据库存储等。

1. 接入集群的性能优化。

云数据库集群接入一般是屏蔽 MySQL 服务器进程，不至于直接暴露在外，通常会有三种功能：路由（包含数据分片的分布）、HA 和防火墙安全。通常接入节点的实现是通过用户态的中间件来实现，复杂点可能会考虑到 SQL 语句解析。这样的话，相比单纯的 MySQL，接入节点通常会损耗 20% 左右的性能。云供应商通常会非常重视接入节点的性能优化，如使用开销更少的开发框架或语言、尽量减少接入节点额外功能等，甚至会把接入功能放到内核态去实现，通过内核网络子系统来完成，尽量降低由于额外增加的接入节点带来的性能损耗。

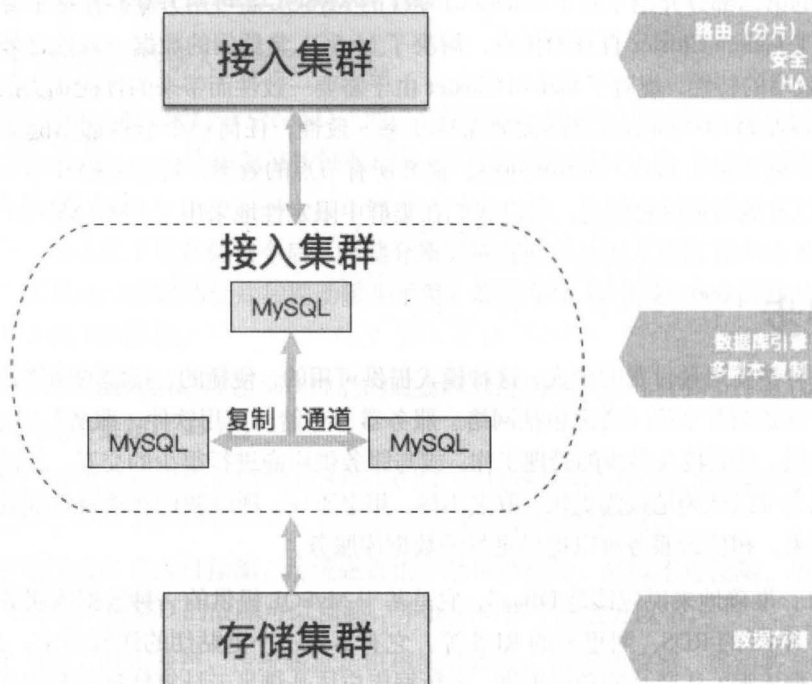


图 26.2

2. 实例集群优化。

实例集群就是通常所说的 MySQL 集群。性能优化一般会包含两个方面：MySQL 引擎本身优化和 MySQL 复制优化。

MySQL 引擎本身优化除了传统 MySQL 性能那些方法之外，最近几年出现了新的情况，就是对 MySQL 内核的优化，如图 26.3 所示。深入到 MySQL 内核优化，既和云数据库远离具体业务相关，也和云数据库自身高速发展相关。目前国内很多团队也在做 MySQL 内核的深度定制，如阿里云的 AliSQL、腾讯云的 TxSQL (Tencent MySQL)。从每次 ACMUG 分享来看，国内 MySQL 内核深度定制能力也逐渐变得很强大。

MySQL 复制优化，也是随着互联网金融等行业的兴起而出现的新方向。在金融行业，业务对数据强一致要求越来越强烈，而 MySQL 官方本身在强一致性下，某些业务场景下的性能损耗高达 50% 以上。这里就如上面提到的，性能和强一致性（或者安全）之间产生了矛盾。为了解决这一矛盾，MySQL 官方也在进步，MySQL Group Replication 就是这种需求的产物，但云计算供应商这块似乎走得更快。目前业界有两种做法：一是对 MySQL Native Replication 进行深入优化，如 Master 的 Binlog Event 发送机制、Slave 的 Binlog Event 接收机制，主要在并行度和锁的效率上进行优化；另外一种就是建立独立的复制通道，

如腾讯云 CDB 的 logbus，它在 Master 和 Slave 之间建立起独立的复制通道来解决性能问题，为了帮助大家理解，附上腾讯云的 logbus 架构简图如图 26.4 所示。这种思路能在降低对 MySQL 内核入侵程度的情况下，快速提升性能。当然，这里的技术挑战也比较大。类似地，还有阿里云的双通道复制，网上已经有很多资料，读者可以自行搜索。

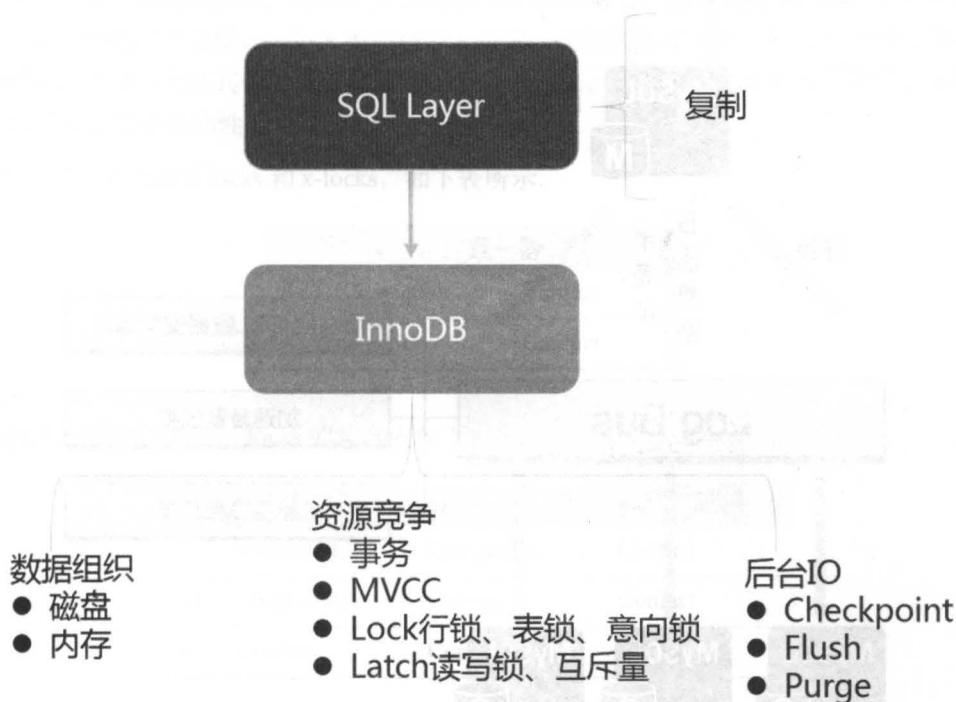
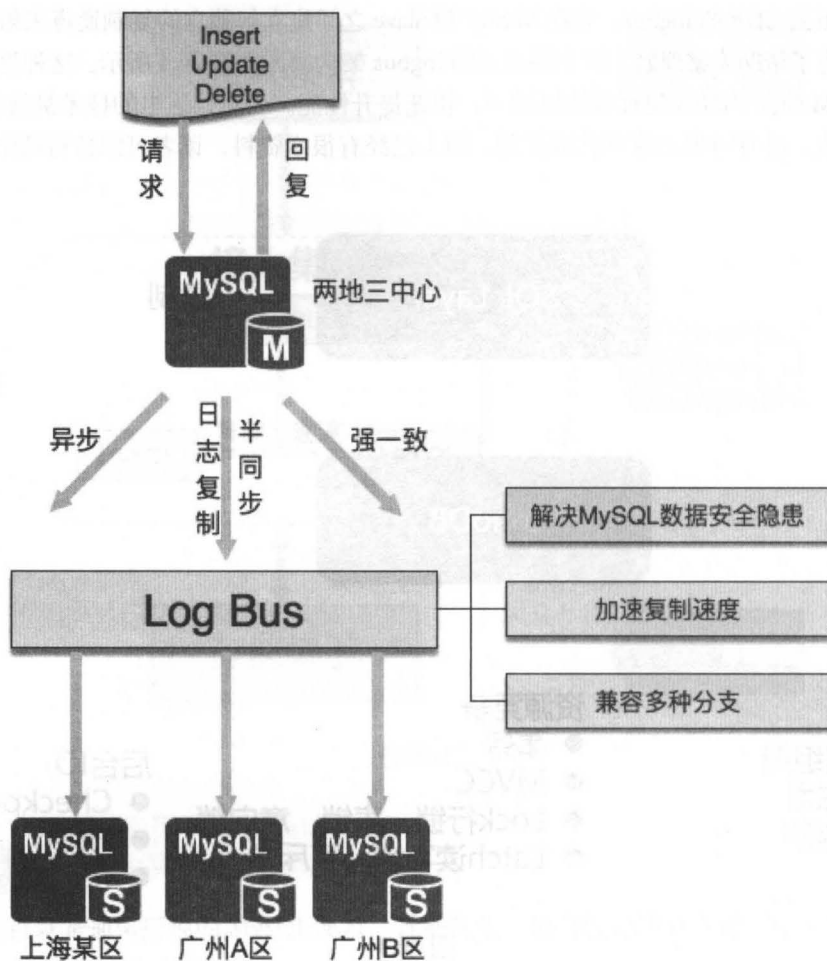


图 26.3

3. 数据库存储优化

目前主流的云数据库存储还是本地存储，本地存储基本上都是使用了 SSD 固态硬盘。针对 SSD 固态硬盘，通常会考虑和供应商一起合作来进行深度定制，比如简化 IO 路径、避免 InnoDB 的 Double Write 等来提升 MySQL 的性能。这里也有比较前瞻的一些产品，如 AWS 的 Aurora（它采取了数据库实例和存储分离的方式，来实现更好的性能和存储容量扩展能力），腾讯云的 CDB 也有类似的版本。这里就需要有一个非常可靠的分布式存储系统来提供存储服务，另外实例之间的数据一致性处理也非常重要。

从传统 MySQL 到云上 MySQL 的演变，云数据库的诞生是符合社会分工精细化的发展趋势，也是人类产能效率进一步提升的必要。从笔者了解到的情况来看，不仅国外的 AWS 和 Oracle 云蓬勃发展，国内腾讯云数据库、阿里云数据库规模也越来越大，在各种功能上，尤其在性能上，也逐渐和传统 MySQL 拉开距离，大家可以考虑去尝试拥抱云数据库产品。



基于Logbus的复制架构

图 26.4

总结

MySQL 性能优化的思路非常繁多，还有很多内容没有介绍。特别是在操作系统和硬件方面，值得优化的地方还有很多，网上的各种资料也有很多，配置也很确定，请自行搜索参考，这里就不浪费篇幅，原样照搬了。

性能优化思路虽多，但条条大路通罗马，最终的目的都是把 MySQL 使用好。不要忘记，MySQL 也是在不断进步的。在本章的最后，祭出一个大招，那就是请升级你的 MySQL 到最新的稳定版本。

MySQL 在最近几年的发展非常快,到目前为止,5.7 版本已经广泛应用于生产环境,而 MySQL 8 版本也已经发布,考察这几个版本的细节就会发现,MySQL 通过内部代码的优化和改造,功能越来越强大,性能也越来越好了。最后举一个例子:在 MySQL 内部使用 rw-lock 锁来保护行级别的修改对象。在 MySQL 5.7 之前,只有 s-locks (共享锁) 和 x-locks (互斥锁) 这两种类型 (不考虑意向锁)。那么在下面的锁交叉调用表中,就会发现 rw-lock 冲突的机会很大。在 MySQL 5.7 之后, sx-lock 类型被引入进来,提供共享互斥锁,在对一个对象加互斥锁的时候,还可以允许其他事务对其进行不一致读操作。这可以优化并行写的能力,大幅度提升读写密集型业务的性能。

MySQL 5.7 之前只有 s-locks 和 x-locks,如下表所示。

	S	X
S	Compatible	Conflict
SX	Conflict	Conflict

MySQL 5.7 之后,加入 sx-locks 的兼容表如下。

	S	SX	X
S	Compatible	Compatible	Conflict
SX	Compatible	Conflict	Conflict
X	Conflict	Conflict	Conflict

类似于上面的优化,是我们在外围无法做到的,需要升级 MySQL 到更高版本才行。有关 MySQL 最新版本的各种功能,请参阅 MySQL 官方在线文档。

本书有关 MySQL 的部分,参考了大量的 MySQL 源代码和 MySQL 官方文档,附上最新官方文档的地址以表达我们最衷心的感谢: <https://dev.mysql.com/doc/refman/5.7/en/>。

27

MySQL Group Replication

本章作者为特邀撰稿人：宋利兵

Group Replication 是 MySQL 官方开发的一个开源插件，是实现 MySQL 高可用集群的一个工具。它的代码包含在 MySQL 的源代码中，二进制插件库也包含在 MySQL 的安装包中。2016 年 12 月 12 日 Group Replication 的第一个 GA 版本正式发布于 MySQL 5.7.17 中。想要使用 Group Replication 的读者，只需要从 MySQL 官方网站中下载并安装 MySQL 5.7.17 及以后的版本即可。

Group Replication 概述

Group Replication 发布以后，有如下 3 种方法来实现 MySQL 的高可用集群。

- 异步复制。
- 半同步复制。
- Group Replication。

异步复制是实现最早也是最简单的高可用方法。相比异步复制而言，半同步复制提高了 MySQL 集群的可靠性。Group Replication 则是 MySQL 复制今后的发展方向。于前两者相比，不仅是可靠性更好，在易用性上也有巨大的提高。为了使用和实现上的简单，Group Replication 大量异步复制了原有的框架和技术。事务仍然会产生 Binlog Event，Group Replication 会将 Binlog Event 发送到其他的 MySQL 服务器上。这些 Binlog Event 也是通过一个内部

自建的 Slave 通道 (Channel) 执行的, 执行过程和异步复制大致相同, 这个通道也支持多线程复制。Binlog Event 产生和执行过程中涉及的参数配置, 大部分也是使用 MySQL 服务器原有的参数进行配置的。因此, Group Replication 是很容易学习和使用的, 凡是使用过 MySQL 异步复制的读者, 都能够很轻松地使用 Group Replication 来搭建 MySQL 复制环境。

2016 年 9 月在 Oracle Openworld 上, MySQL 官方提出了 MySQL InnoDB Cluster 的概念, 并将其定为 MySQL 未来的发展目标。其核心就是用 MySQL 数据库和原生工具构建出一个全栈高可用的 MySQL 集群系统, 用来支撑大规模 MySQL 集群的使用, 特别是在云上的使用。除了全栈高可用的特点外, 易用性也是 MySQL InnoDB Cluster 的一个重要特征。MySQL 官方的期望是: 用户在使用 MySQL InnoDB Cluster 时只需要几个人而不是一整个团队来维护庞大数量的 MySQL 数据库系统。Group Replication 是 MySQL InnoDB Cluster 这个极具潜力产品中的一个核心组件, 是实现 MySQL InnoDB Cluster 高可用性和易用性的基础。

组的概念

Group Replication 插件中有组 (Group) 的概念, 被 Group Replication 插件连接在一起的 MySQL 服务器是一个高可用组, 组内的 MySQL 服务器被称为成员 (Member)。组的概念贯穿于 Group Replication 的使用和内部实现之中。Group Replication 内部集成了组管理服务, 实现了很多组内成员的自动化管理功能, 这使得 Group Replication 的使用和管理变得非常简单。当用户使用 Group Replication 插件搭建复制环境时, 就是在管理 Group Replication 组。用户对 Group Replication 组的管理有三种操作, 分别如下。

- 创建组: 当组的第一个成员启动时, 需要对组进行初始化。
- 加入组: 将 MySQL 服务器加入到一个存在的 Group Replication 组内。
- 离开组: 从一个 Group Replication 组内移除一台 MySQL 服务器。

Group Replication 中, 用户不再需要通过原有的 Slave 命令 CHANGE MASTER、START SLAVE、STOP SLAVE 来控制通道。当组初始化后, 组的第一个成员会自动成为 Master。新加入的成员会自动从组内的 Master 上复制数据。这些使用到的通道由 Group Replication 插件自动控制, 不需要用户的干预。特别是当 MySQL 服务器出现故障需要做切换时, 不再需要做 STOP SLAVE、CHANGE MASTER、START SLAVE 这样的操作, 选择新的 Master 之后, 整个切换过程会自动完成。因此通过 Group Replication 搭建 MySQL 高可用系统变得非常简单, 只需要使用一个非常简单的 Failover 工具, 甚至不使用额外的 Failover 工具, 就能够让 MySQL 高可用集群运行起来。

多主复制

Group Replication 支持像异步/半同步复制一样可以做一主多从的复制, Group Replication 里称作单主复制。除此之外, Group Replication 还提供了一种更高级的复制模式,叫作多主复制。在多主模式下,所有成员同时对外提供的读写服务都是 Master,彼此之间会自动进行数据复制。这是一种真正意义上的多主并发复制,用户可以像在一个 MySQL 服务器上更新数据一样,并发地在多个成员上更新数据。Group Replication 插件能够将这些并发事务的更新操作同步到每个成员上,使它们的数据保持一致。多个成员上的并发更新有冲突时(比如修改了同一条记录),Group Replication 能够检测出这些冲突并做出正确的处理。多主模式提供了很好的高可用性和易用性,使得 Group Replication 可以适应更多的使用场景。多主复制能提供以下额外的优势。

- 当一个成员发生故障时,只会造成一部分连接失效,对应用程序的影响会小一些。
- 当需要关闭某个 MySQL 服务器时,可以先将其上的连接平滑地转移到其他的成员上后再关闭这个成员,不会造成应用的瞬时中断。
- 多主模式的性能很好,对瞬时的高并发有着很好的承载能力。

单独的通信机制

虽然 Group Replication 会使用 Slave 的通道,但只是使用这个通道的执行线程 (Applier Thread) 来执行 Binlog Event,并没有使用这个通道来传输 Binlog Event。Group Replication 不但不使用异步复制的 Binlog Event 传输机制,也不使用 MySQL 的服务端口来进行通信。Group Replication 创建了一个独立的 TCP 端口进行通信,各个 MySQL 服务器上的 Group Replication 插件通过这个端口连接在一起,两两之间可以直接通信,如图 27.1 所示。

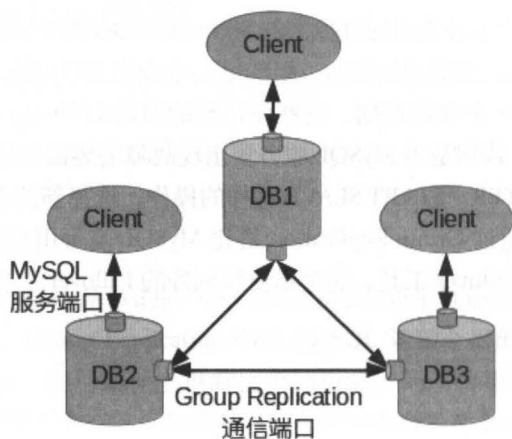


图 27.1

Binlog Event 的传输并不像异步复制那样是简单的点到点之间的传输。Group Replication 在传输数据时,使用了 Paxos 协议。Paxos 协议保证了数据传输的一致性和原子性。Group Replication 基于 Paxos 协议构建了一个分布式的状态机复制机制,这是实现多主复制的核心技术。这个技术为 Group Replication 带来了三个主要优点,分别如下。

- Group Replication 中不会出现脑裂的现象。
- Group Replication 的冗余能力很好,能够保证 Binlog Event 至少被复制到超过一半的成员上,只要同时宕机的成员不超过半数便不会导致数据丢失。
- Group Replication 还保证只要 Binlog Event 没有被传输到半数以上的成员,本地成员不会将事务的 Binlog Event 写入 Binlog 文件和提交事务,从而保证宕机的服务器上不会有组内在线成员上不存在的数据库。因此,宕机的服务器重启后,不再需要特殊的处理就可以加入组。

Group Replication 服务模式

Group Replication 组对外提供服务的时候有 2 种服务模式,分别如下。

- 单主模式。
- 多主模式。

单主模式

单主模式下只有一个成员提供更新服务,其他成员只提供查询服务。提供更新服务的成员叫作主成员 (Primary Member),只提供查询服务的叫作从成员 (Secondary Member)。Group Replication 的单主模式是异步复制和半同步复制的替代方案。下面来了解一下单主复制模式的使用特点。

主成员的自动选取和切换

单主模式下,组内的成员会自动选举出主成员。初始化时,被初始化的成员自动选举为主成员,其他加入组的成员自动成为从成员。当主成员发生故障或离开组时,会从组内的其他成员中选出一个新的主成员。选取主成员的方法很简单,首先对所有在线成员的 UUID 进行排序,然后选取 UUID 最小的成员作为主成员。Group Replication 插件中实现了一个 MySQL 状态变量,用来向用户显示组内当前主成员的 UUID。用户在任何一个在线的成员上都能查询到当前主成员的 UUID,如下所示。


```
SHOW GLOBAL STATUS LIKE "group_replication_primary_member";
```

用户也可以通过 SQL 语句从 performance_schema 的 global_status 表中查询到主成员的 UUID，如下所示。

```
SELECT * FROM performance_schema.global_status  
WHERE variable_name = 'group_replication_primary_member';
```

读写模式的自动切换

当一个成员加入组时，Group Replication 插件会自动将 MySQL 变成只读模式，只有被选取为主成员后才会自动切换回读/写模式。对 MySQL 只读模式的控制是通过下面的 SQL 语句进行的。

```
SET GLOBAL super_read_only = 1;  
SET GLOBAL super_read_only = 0;
```

Failover

当主成员故障时，组内会自动选出新的主成员，复制也能正常进行。因此组内的 Failover 是完全自动化的，不需要用户干预。而客户端（应用程序或中间件）要做的事情，就是从其他成员上查询到主成员的 UUID，然后将连接迁移到新的主成员上。这一点和异步复制不一样，因此在现有的系统中使用 Group Replication 时，需要对应用程序或中间件做一点改动。

多主模式

多主模式下，组中所有的成员同时对外提供查询和更新服务，且没有主从之分，成员之间是完全对等的。客户端连接到任何一个成员上，都能进行读/写操作，就好像在操作同一个 MySQL 服务器。

自增字段的处理

当使用多主模式时，需要设置 autoincrement 相关的参数来保证自增字段在每个成员上产生不同的值。Group Replication 提供了两种配置方式，分别如下。

- 直接配置 MySQL 的系统变量。通过命令来配置时，需要在所有的成员上配置下面两个参数。

```
SET GLOBAL auto_increment_offset= N;  
SET GLOBAL auto_increment_increment= N;
```

- 通过 Group Replication 插件来配置。Group Replication 插件定义了一个新的参数用来配置自增字段的大小，如下所示。

 SET GLOBAL group_replication_auto_increment_increment = N;

group_replication_auto_increment_increment 的默认值是 7。如果自增值的浪费不对业务造成影响，可以不用修改这个值。Group Replication 没有提供新的参数来设置自增值的偏移，而是将 MySQL 服务器的 server-id 看作自增值的偏移。如果用户的 server-id 本来就是按照 1、2、3 这样的顺序配置的，就不需要再做额外的配置。在启动 Group Replication 插件时，它会检测用户是否配置了 MySQL 的自增变量。如果用户没有配置这两个变量（auto_increment_offset 和 auto_increment_increment 都为 1），则会自动将 group_replication_auto_increment_increment 和 server-id 的值设置到 MySQL 的 auto_increment_increment 和 auto_increment_offset 全局变量中。



注意：一些用户在实践中不使用 1、2、3 这样顺序的值来配置 server_id 时，就需要手动来配置 MySQL 的自增变量。

自增变量将从 1 开始的连续自增值划分为固定大小的段，各个 MySQL 服务器分别使用段内不同偏移上的自增值。auto_increment_increment 代表段的大小，每个成员都应该配置相同的段大小。而 auto_increment_offset 是本成员的自增值在段内的偏移。每个成员应该设置不同的偏移量。假设段的大小设为 5，成员 A 的偏移量设为 1，成员 B 的偏移量设为 2，那么成员 A 上产生的自增值为：1、6、11、16……，成员 B 上产生的自增值为：2、7、12、17……。如图 27.2 所示。

	偏移 1	偏移 2	偏移 3	偏移 4	偏移 5
第一段	1	2	3	4	5
第二段	6	7	8	9	10
第三段	11	12	13	14	15
第四段	16	17	18	19	20
...

图 27.2

可以看出，自增字段的大小依赖于 Group Replication 组中成员的多少。auto_increment_increment 最小要等于 Group Replication 组内成员的数量。如果段的大小等于组内成员的数量，则所有的自增值都会被使用。如果段的大小大于组内成员的数量，则有一部分自增值永远不会被使用，会造成一定的浪费。比如，组内有 3 个成员，偏移分别是 1、2、3，而段的大小是 5，那么偏移 4 和 5 上的值就会被浪费掉。考虑到 Group Replication 组的扩展，最好将段的大小设置得比现有的组内成员数量大一些。这样虽然会浪费掉一些自增值，但是扩展时会很简单。在使用的过程中变更自增字段的大小，处理起来会比较麻烦，所以要提前规划好。

多主模式的限制

受制于 MySQL 现有的技术特点,多主模式在使用时还有一些限制。

- 不支持串行 (SERIALIZABLE) 的隔离级别。单个 MySQL 服务器中,通过锁的方式来实现串行化的隔离级别。而多主模式时,多个成员之间的并发操作无法通过锁来实现串行的隔离级别 (至少目前还不能)。
- 不支持外键的级联操作

Group Replication 插件提供了下面这个变量来控制是否做以上限制的检测。如果开启了这个参数,当发现这些情况时就会报错。

```
group_replication_enforce_update_everywhere_checks = TRUE
```

DDL 语句并发执行的问题

多主复制时,通过冲突检测来辨别有冲突的事务,有冲突的事务通过回滚操作来处理。MySQL 5.7 上的 DDL 不是原子操作无法回滚,因此 Group Replication 没有对 DDL 做冲突检测。换句话说,DDL 语句不会和其他任何语句冲突 (包括 DML 和 DDL)。如果 DDL 和有冲突的语句在不同的成员上同时执行,可能导致错误或数据不一致。假设在成员 A 上执行如下事务:

```
BEGIN;  
INSERT INTO t1 VALUES(1);
```

与此同时,成员 B 上执行以下语句:

```
TRUNCATE t1;
```

成员 A 上接着提交事务:

```
COMMIT;
```

成员 A 上两个事务的执行顺序将是:

```
INSERT INTO t1 VALUES(1);  
TRUNCATE t1;
```

成员 B 上两个事务的执行顺序将是:

```
TRUNCATE t1;  
INSERT INTO t1 VALUES(1);
```

这就会导致两个成员上数据的不一致。因此多主模式下,执行任何 DDL 前,要将可能有冲突的事务迁移到同一台 MySQL 服务器上,再开始执行 DDL 语句。

对多主模式在使用上的一些思考

虽然多主模式有以上的一些限制,但优点也很突出,还是很值得尝试的。使用多主模式有两个额外的条件,分别如下。

- 应用或中间件要能够把写请求分发到多个成员上。
- 要能够控制 DDL 的使用。当有 DDL 要执行时,能够把所有的写请求转移到同一台 MySQL 上去执行。

如果直接使用多主模式比较困难,还有一个折中的使用方式可以考虑,就是把多主当作主从的复制来使用。

- 从 MySQL 的层面来看, Group Replication 组内的 MySQL 服务器的状态和异步复制环境中的 MySQL 服务器的状态是没有区别的,可以直接在原来的高可用环境中使用。对于原有 Failover 工具的改造,只是去掉主从切换的过程就可以了。
- 由于原有的主从复制本身就控制写操作在同一台服务器上执行,不会出现 DDL 和其他语句的冲突问题。因此把多主当作主从复制使用是安全的,而且还能避免原有环境中可能出现的脑裂问题。因为在多个 Group Replication 成员上同时写时不会导致问题,除非刚好碰见 DDL 的冲突,而 DDL 的冲突在原有的主从环境发生脑裂时也会发生。
- 如果这个 MySQL 集群为多个应用提供服务并且这些应用之间不会更新同样的数据,可以将不同应用的写操作分布到不同的成员上。这会提高 MySQL 集群的性能。

服务模式的配置

服务模式的配置通过下面的变量进行。

```
 SET GLOBAL group_replication_single_primary_mode = OFF;
```

单主模式和异步复制非常相似,十分容易上手,而且单主模式在使用上也没有什么限制。因此, Group Replication 将单主模式设为了默认模式。如果要使用多主模式,则需要在加入组前将这个变量设置为 OFF。服务模式是不能在线切换的,必须使组内的所有成员退出组,然后重新初始化组为要使用的服务模式,再把其他成员加入进来。

Binlog Event 的多线程执行

group_replication_applier 通道

Group Replication 插件会自动创建一个通道(Channel)来执行接收到的 Binlog Event,通道的名字是 group_replication_applier。当加入组时, Group Replication 插件会自动启动 group_replication_applier 通道的执行线程 (Applier Thread)。如果用户需要调整 group_replication_applier

执行线程的参数,也可以手动停止和启动这个通道的执行线程,操作命令如下所示。

```
START SLAVE SQL_THREAD FOR CHANNEL 'group_replication_applier';  
STOP SLAVE SQL_THREAD FOR CHANNEL 'group_replication_applier';
```

基于主键的并行执行

Group Replication 中的 Binlog Event 的执行也支持多线程并行执行,配置方法如下。

```
SET GLOBAL slave_parallel_type = 'LOGICAL_CLOCK';  
SET GLOBAL slave_parallel_workers = N;  
SET GLOBAL slave_preserve_commit_order = ON;
```

开启并行复制,需要设置并行复制的类型为 LOGICAL_CLOCK。不过,Group Replication 的并行复制算法和异步复制中的 LOGICAL_CLOCK 算法并不相同。Group Replication 并发策略中的逻辑时间是基于主键计算出来的,比异步复制基于锁计算出来的逻辑时间的并发性能要好很多。基于主键的并行复制有以下两个特点。

- 如果两个事务更新了同一行数据,则按顺序执行;否则,就可以并发执行。
- DDL 不能和任何事务并发执行,必须等待它前面的所有事务执行完毕后才能开始执行。后面的事务也必须等待 DDL 执行完毕后,才能开始执行。

并发执行的时候,不管两个事务的 Binlog Event 是不是同一个 Session 产生的,只要满足上面的特点就能够并发执行。因此,同一个 Session 里的事务也可能被安排并发执行,这就会导致后执行的事务先被提交的情况。例如一个 Session 中连续执行了下面两个事务。

```
UPDATE t1 SET c2 = 5 WHERE pk = 1;  
INSERT INTO t1(pk, c2) VALUES(2, 5);
```

由于两个事务没有更新同一行数据,所以它们的 Binlog Event 会被安排并发执行,这样就有可能出现 INSERT 比 UPDATE 先提交的情况。为了保证同一个 Session 中的事务按照同样的顺序提交,Group Replication 在开启并行复制时,要求必须要设置 slave_preserve_commit_order 的值为 ON。打开这个参数可以保证 Applier 上执行事务的提交顺序和源 MySQL 服务器上的提交顺序相同。

搭建 Group Replication 复制环境

本节向大家介绍搭建 Group Replication 复制环境所涉及的基本配置命令和操作步骤。操作中会使用到一些 MySQL 异步复制的配置参数,相信使用过的人都很熟悉,这里不做重点介绍。本节重点介绍 Group Replication 的一些概念,以及操作过程中可能会遇到的问题。

MySQL 的参数设置

Group Replication 需要使用 Binlog、Slave 的一些特性，因此在启用 Group Replication 插件之前，要先对 MySQL 做一些配置工作。

开启 Binlog 和 Relaylog 功能

需要设置以下参数。

```
server_id = 1
log_bin = binlog
log_slave_updates = ON
relay_log = relay-log
```

开启 GTID 功能

Group Replication 要求必须使用 GTID 功能。

```
gtid_mode = ON
enforce_gtid_consistency = ON
```

设置 Row 格式的 Binlog

Group Replication 只支持 Row 格式的 Binlog。

```
binlog_format = ROW
```

禁用 binlog_checksum

Group Replication 目前 (MySQL 5.7.17) 还不支持带 Checksum 的 Binlog Event。

```
binlog_checksum = NONE
```

使用系统表来存储 Slave 的信息

Group Replication 要使用到多源复制的功能，多源复制要求必须将 Slave 通道 (Channel) 的状态信息存储到系统表中。

```
master_info_repository = TABLE
relay_log_info_repository = TABLE
```

开启并行复制

```
SET GLOBAL slave_parallel_type = 'LOGICAL_CLOCK';  
SET GLOBAL slave_parallel_workers = <线程数量>;  
SET GLOBAL slave_preserve_commit_order = ON;
```

开启主键信息采集功能

除了使用 Binlog 之外, Group Replication 还需要 Server 层帮助采集被更新数据的主键信息。以下参数用来告诉 Server 层是否采集主键信息。

```
transaction_write_set_extraction = XXHASH64;
```

主键信息被哈希后存储起来, 目前支持两种哈希算法: XXHASH64 和 MURMUR32。这个参数是专门给 Group Replication 准备的, 默认情况下为 OFF, 即不采集主键信息。当它的值被设置成 XXHASH64 或 MURMUR32 时, 就会采集主键信息。主键信息是 Group Replication 中非常重要的信息, 因此 Group Replication 要求每个表必须要有主键, 否则在更新数据时就会失败。



注意: 一个组内的所有成员上必须要配置同样的哈希算法。

Group Replication 插件的使用

加载插件

通过 MySQL 的插件加载命令来加载 Group Replication 插件。

```
INSTALL PLUGIN group_replication SONAME 'group_replication.so';
```

启用插件

通过下面的命令启用 Group Replication 插件。

```
START GROUP_REPLICATION;
```

这个命令将本 MySQL 服务器加入到一个存在的 Group Replication 组内, 或者将它初始化为组内的第一个成员。

停用插件

通过下面的命令停用 Group Replication 插件。

```
STOP GROUP_REPLICATION;
```

这个命令将 MySQL 服务器从一个 Group Replication 组内移除出去。

Group Replication 插件的基本参数设置

设置组的名字

每个 Group Replication 都需要有名字，这个名字唯一识别一个组。在初始化或加入一个组时，必须要设置组的名字。成员加入组时会校验设置的组名是否和正在加入的组同名。如果名字不同，就不能加入。

```
SET GLOBAL group_replication_group_name = <a uuid>;
```

名字要求必须是一个 UUID。这个 UUID 会用来标记组内所有成员上产生的 Binlog Event，任何成员上生成的 GTID 都会使用这个 UUID。这样很容易分辨出 Binlog Event 是哪个组产生的。

设置成员的本地地址

每个成员都要有一个独立的 TCP 端口，成员之间通过这个端口进行通信。成员地址包括 IP 地址和 TCP 端口两部分。

```
SET group_replication_local_address = <ip:port>;
```

设置种子成员的地址

当加入一个组时，新成员首先必须要和组内的成员进行通信来完成加入的步骤，因此需要知道至少一个当前组内成员的地址。这些成员称为种子成员，通过下面这个参数配置。

```
SET GLOBAL group_replication_group_seeds = <ip:port,ip:port>;
```

这个变量里配置的是其他成员的 `group_replication_local_address` 的内容。

设置成员 IP 白名单

Group Replication 通过白名单来控制哪些 IP 地址的 MySQL 服务器能够加入到组里来。白名单的值是一个地址列表，地址可以是具体 IP，也可以是网段。地址之间用“,”分隔。

```
SET group_replication_whitelist = <ip,net,...>;
```


如果用户不配置白名单，Group Replication 插件会自动识别本机网口上配置的私网地址和私网网段，只允许和自己在同私网网段的 MySQL 服务器连接到自己的端口。127.0.0.1 的连

接请求始终都是被允许的。从安全的角度来考虑,建议不要使用默认配置,也不要将整个网段加入白名单,而是将成员的 IP 地址逐个加入白名单中。配置白名单必须要关闭 Group Replication,因此建议提前规划好所有的 IP 地址,一次性设置完成,从而避免以后不必要的重启。另外,Group Replication 还支持 SSL 通信,MySQL 使用手册上有详细的介绍,有需要的可以参考手册。

手册地址: <https://dev.mysql.com/doc/refman/5.7/en/group-replication-secure-socket-layer-support-ssl.html>。

将参数写入配置文件

插件的参数只能在插件加载之后设置。如果想设置这些参数到配置文件中,可以在参数前加上“loose-”前缀。

```
 loose-group_replication_group_name = <a uuid>;  
loose-group_replication_local_address = '127.0.0.1:12345'
```

Group Replication 的数据库用户

_gr_user@localhost 用户

Group Replication 中会使用一些 SQL 语句来查询和配置数据库中的一些参数。这些 SQL 语句是通过一个 MySQL Session 执行的。执行这些语句时,使用的用户叫作 _gr_user@localhost。这个用户是在加载 Group Replication 时创建的,创建之前会检查这个用户是否存在。


root 用户

当检查 _gr_user@localhost 用户是否存在和创建 _gr_user@localhost 用户时,这个 Session 使用的是 root 用户。有些用户在实践中会把 root 用户删除掉,在使用 Group Replication 时不能这么做,必须要有 root 用户。

Group Replication 组初始化

初始化特有的参数

Group Replication 组的初始化是在启用第一个成员时完成的。在启用第一个成员时,需要设置下面的参数告诉 Group Replication 插件:它是该组的第一个成员,需要做一些初始化工作。

```
 SET GLOBAL group_replication_bootstrap_group = ON;
```



注意：这个参数只在初始化第一个成员时使用，所以不要将这个参数设置到配置文件中，并且在初始化完成后要将该变量设置成 OFF。

组初始化的步骤

MySQL 的相关配置略过不提，Group Replication 组的初始化步骤如下。

```
INSTALL PLUGIN group_replication SONAME "group_replication.so";
SET GLOBAL group_replication_group_name = "12345678-1234-1234-1234-1234567890ab";
SET GLOBAL group_replication_local_address = "127.0.0.1:20001";
SET GLOBAL group_replication_bootstrap_group = ON;
START GROUP REPLICATION;
SET GLOBAL group_replication_bootstrap_group = OFF;
```

新成员加入组

配置 group_replication_recovery 通道

当新成员加入一个 Group Replication 组后，首先要从其他节点上把它加入之前的数据复制过来。这些以前的数据不能通过 Group Replication 的通信协议进行复制，而是使用了异步复制的机制。Group Replication 需要使用一个名叫 group_replication_recovery 的异步复制通道 (Channel)。用户必须要提前配置这个通道，不过只需要为这个通道配置连接需要的用户名和密码，其他参数在 Group Replication 插件启动 group_replication_recovery 通道时会自动进行配置。

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass'
FOR CHANNEL 'group_replication_recovery';
```



注意：连接到哪个成员上去复制数据，是由 Group Replication 插件随机选择的，因此为 group_replication_recovery 配置的用户要在每一个成员上存在。

在启动 group_replication_recovery 通道之前，Group Replication 会自动为其配置 MASTER_HOST 和 MASTER_PORT。当一个成员加入组时，会收到组内其他成员的配置信息。配置信息中包括主机名和 MySQL 服务端口号。主机名和端口号是从全局只读变量 hostname 和 port 中获取的。如果 hostname 无法正常解析成 IP 地址或网络中使用了网络地址映射，group_replication_recovery 通道就无法正常工作。有如下两种办法来解决这个问题。

- 在 /etc/hosts 中配置所有成员的主机名和 IP 地址的对应关系。

- 配置 MySQL 的 `report_host` 和 `report_port` 的命令行参数。如果用户配置了 `report_host` 或 `report_port`, 那么 Group Replication 会优先使用这个变量中的地址和端口。

成员加入组的步骤

MySQL 的相关配置和创建复制用户的步骤略过不提, Group Replication 加入组的步骤如下。

```
INSTALL PLUGIN group_replication SONAME 'group_replication.so';
SET GLOBAL group_replication_group_name = "12345678-1234-1234-1234-1234567890ab";
SET GLOBAL group_replication_local_address = "127.0.0.1:20002";
SET GLOBAL group_replication_group_seeds = "127.0.0.1:20001";
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass'
    FOR CHANNEL 'group_replication_recovery';
START GROUP_REPLICATION;
```

可以看到, Group Replication 的配置过程还是非常简单的。

Group Replication 的高可用性

Group Replication 数据传输使用的 Paxos 协议是一个“多数派”的协议,任何数据的传输都要收到超过一半成员的应答才算成功。为了数据能够正常传输,要求至少有半数以上的成员能够正常通信。Group Replication 组最多能有 9 个成员,不同大小的组的冗余能力如图 27.3 所示。

成员数量	半数以上	冗余能力
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

图 27.3

当组内只有两个成员时, Group Replication 没有冗余能力。任何一个成员的故障都会导致整个组不可用,单主、多主模式都如此。这一点和异步复制不一样,所以在搭建 Group Replication

时,至少要有三个成员才能有冗余能力。另外,组的成员数为奇数个或奇数+1个时,冗余能力是相同的。通常情况下,搭建 Group Replication 组时,采用奇数个成员数量就可以了。

组内成员数量的变化

组内成员数量是自动计算的,当用户通过 START GROUP_REPLICATION 将一个成员加入组时,组内成员总数会自动增加。当用户通过 STOP GROUP_REPLICATION 让一个成员离开组时,组内成员的数量会自动减少。这两种操作都是用户对组的正常维护操作,会导致组的大小变化。当网络问题或故障导致成员不可用时,是不会引起组的变化的。因为这不是正常的组管理操作,是用户不期望的变化。

强制移除故障成员

当故障导致半数以上的成员不可用时,Group Replication 就不能对外提供服务。这种情况是用户不希望看到的,但偶尔也会发生。当这种情况发生时,用户会希望在线的少数成员能够立刻对外提供服务。虽然冗余能力不好,但至少能保障业务不中断。Group Replication 提供了一个参数来实现这个功能。



```
SET GLOBAL group_replication_force_members = <ip:port,ip:port>;
```

group_replication_force_members 中配置的是成员地址的列表,只需要在列表中的任意一个成员上设置即可。这个变量强制 Group Replication 用参数中的几个成员来构成组,把其他成员从组内移除出去。



注意:这个操作打破了“多数派”协议的基本原理,所以在使用这个参数前,一定要确保组内其他的 MySQL 服务器不会对外提供服务,否则可能导致脑裂。

Group Replication 的监控

Group Replication 的状态信息被存储到了以下五个 performance_schema 表中,可以很方便地进行 SQL 语句查询。

- replication_group_members。
- replication_group_member_stats。
- replication_connection_status。
- replication_applier_status。
- threads。

replication_group_members

replication_group_members 中存储着组内的所有成员的基本信息,从任何一个成员上都能查询到这些基本信息。replication_group_members 包含的字段如下表所示。

字段	说明
CHANNEL_NAME	Group Replication 执行 Binlog Event 的通道,这个字段的值是“group_replication_applier”
MEMBER_ID	成员的 UUID,和每个成员上 MySQL 全局变量 server_uuid 的内容相同
MEMBER_HOST	成员的 hostname,和每个成员上 MySQL 全局变量 hostname 或 report_host 的内容相同
MEMBER_PORT	成员的 MySQL 服务端口,和每个成员上 MySQL 全局变量 port 或 report_port 的内容相同
MEMBER_STATE	成员的状态

成员的状态有以下五种。

- OFFLINE: 当 MySQL 服务器的 Group Replication 插件没有启动时,状态为 OFFLINE。
- RECOVERING: 当 MySQL 服务器的 Group Replication 插件启动后,会首先设置成 RECOVERING 状态,开始复制加入前的数据。
- ONLINE: 当 Recovery 过程完成后,状态设置为 ONLINE,开始对外提供服务。
- ERROR: 当本地成员上发生错误,Group Replication 无法正常进行时,当前成员的状态会变成错误。
- UNREACHABLE: 当网络故障或其他成员宕机时,其他成员的状态会被设置为 UNREACHABLE。

replication_group_member_stats

replication_group_member_stats 中存储着本地成员的详细信息,每个成员上只能查询到自己的详细信息。replication_group_member_stats 包括的字段如下表所示。

字段	说明
CHANNEL_NAME	Group Replication 执行 Binlog Event 的 channel 通道,这个字段值是“group_replication_applier”

续表

字段	说明
MEMBER_ID	成员的 UUID, 和每个成员上 MySQL 全局变量 <code>server_uuid</code> 的内容相同
VIEW_ID	组视图的 ID
COUNT_TRANSACTIONS_IN_QUEUE	队列中等待做全局事务认证的事务数量
COUNT_TRANSACTIONS_CHECKED	做了全局事务认证的事务总数量, 从加入组开始累计
COUNT_CONFLICTS_DETECTED	全局事务认证时, 有冲突的事务的总数量
COUNT_TRANSACTIONS_VALIDATING	冲突检测数据库的记录总行数
TRANSACTION_COMMITTED_ALL_MEMBERS	在所有成员上已经执行了的事务的 GTID 集合。相当于所有成员的“ <code>gtid_executed</code> ”的交集。不是实时的, 每隔一段时间更新一次
LAST_CONFLICT_FREE_TRANSACTION	最后一个没有冲突的事务的 GTID

replication_connection_status

当 MySQL 服务器加入一个组后, 首先要通过异步复制的通道 (Channel) `group_replication_recovery` 把加入组之前组内产生的数据复制过来, 这个通道的状态信息和其他异步复制的通道一样可以通过 `replication_connection_status` 表进行监控。Group Replication 的通道不会在 `SHOW SLAVE STATUS` 的结果中显示。

replication_applier_status

Group Replication 通过 `group_replication_applier` 通道来执行 Binlog Event。 `group_replication_applier` 的状态信息和其他异步复制通道一样可以通过 `replication_applier_status` 进行查询。

threads

Group Replication 线程的状态信息可以通过 `threads` 表进行查询。目前可以查询到 Group Replication 线程分别如下。

- `thread/group_rpl/THD_applier_module_receiver`。
- `thread/group_rpl/THD_certifier_broadcast`。
- `thread/group_rpl/THD_recovery`。

本节中出现了很多新名字，特别是 replication_group_member_stats 表中的字段名。这些名字大多数和多主模式的冲突检测相关。后面的章节会对 Group Replication 所使用的技术做简要介绍，对理解这些名字有很大的帮助。

Group Replication 的基本原理

状态机复制

本质上来说，Group Replication 是一个状态机复制的集群。在状态机复制的架构中，数据库被当作一个状态机。每一次写操作都会导致数据库的状态变化。为了创建一个高可用的数据库集群，有一个组件将这些操作按照同样的顺序发送到多个初始状态一致的数据库上，让这些数据库执行同样的操作。因为初始状态相同，每次执行的操作也相同，所以每次状态变化后各个数据库服务器上的数据保持一致。图 27.4 是一个非常简单的状态机复制的示意图。

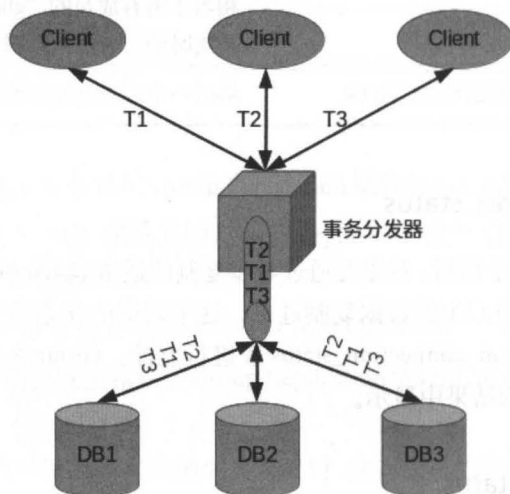


图 27.4

假设图 27.4 中数据库 DB1、DB2 和 DB3 的初始状态是相同的，事务 T1、T2 和 T3 分别如下。

T1

```
DELETE FROM t1 WHERE pk = 1;
```

T2

```
UPDATE t1 SET c1 = 100 WHERE pk = 1;
```

T3

```
UPDATE t1 SET c1 = 50 WHERE pk = 1;
```

图 27.4 中的 3 个客户端并发地将这 3 个事务发送到事务分发器上，事务分发器首先对这 3 个事务进行排序，然后按照同样的顺序（T3、T1、T2）并发地发送到 DB1、DB2 和 DB3 这 3 个数据库上去执行。T3 最先在 DB1、DB2 和 DB3 上执行，执行完后，DB1、DB2 和 DB3 上的结果相同。接着 T1 被执行，执行后 DB1、DB2 和 DB3 上 t1 表中 pk 为 1 的记录都被删除了。最后 T2 被执行，由于 T1 删除了记录，因此 T2 执行时会失败。T2 在 DB1、DB2 和 DB3 上都会失败，所以 DB1、DB2 和 DB3 的数据仍然是一致的。

分布式的状态机复制

在图 27.4 的模型中，事务分发器是一个单点故障点。为了避免单点故障，可以采用分布式的状态机复制。在分布式的状态机复制中，有多个事务分发器，它们彼此相互通信。事务分发器可以同时接收事务请求，就像单个事务分发器同时接收事务请求一样。从应用层面来说，并发的事务发到同一个事务分发器和发到不同的事务分发器上效果是一样的。事务分发器之间会互相通信，把所有的事务汇总、排序。最终，每个事务分发器上都有一份完整的排序的事务请求。每个事务分发器只连接到一台数据库服务器上，并负责把事务请求依次发送到这个相连的数据库上去执行。图 27.5 所示的是分布式状态机复制的一个模型。

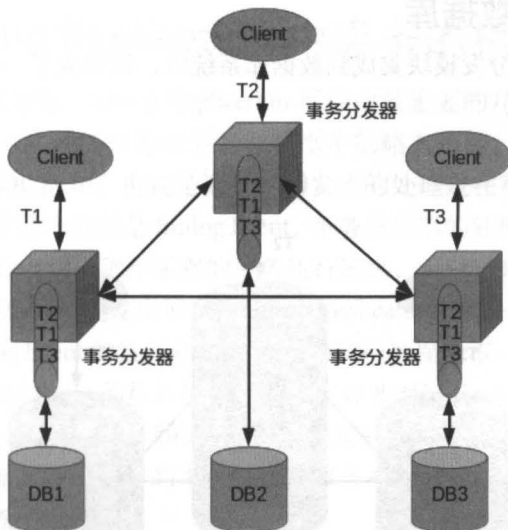


图 27.5

假设图 27.5 中数据库 DB1、DB2 和 DB3 的初始状态是相同的，事务 T1、T2 和 T3 分别如下。



```
# T1
DELETE FROM t1 WHERE pk = 1;
```


T2

UPDATE t1 SET c1 = 100 WHERE pk = 1;

T3

UPDATE t1 SET c1 = 50 WHERE pk = 1;

图 27.5 中的三个客户端并发地将这三个事务发送到各自的事务分发器上。事务分发器彼此之间互相通信, 最终每个事务分发器上都会有一份完整的顺序一样的事务请求列表 (T3、T1、T2)。三个事务分发器将这些事务请求按照顺序发送到各自的数据库服务器上去执行。T3 最先在 DB1、DB2 和 DB3 上执行, 执行完后 DB1、DB2 和 DB3 上的结果相同。接着 T1 被执行, 执行后 DB1、DB2 和 DB3 上 t1 表中 pk 为 1 的记录都被删除了。最后 T2 被执行, 由于 T1 删除了记录, T2 执行时会失败。T2 在 DB1、DB2 和 DB3 上都会失败, 所以 DB1、DB2 和 DB3 的数据仍然是一致的。

不管是单个事务分发器还是多个事务分发器协同工作, 它们的目标都是一样的: 将所有的事务按同样的顺序发送到数据库服务器上去执行。

分布式的高可用数据库

如果将图 27.5 中的事务分发模块集成到数据库系统中, 就变成了一个分布式的高可用数据库系统, 如图 27.6 所示。

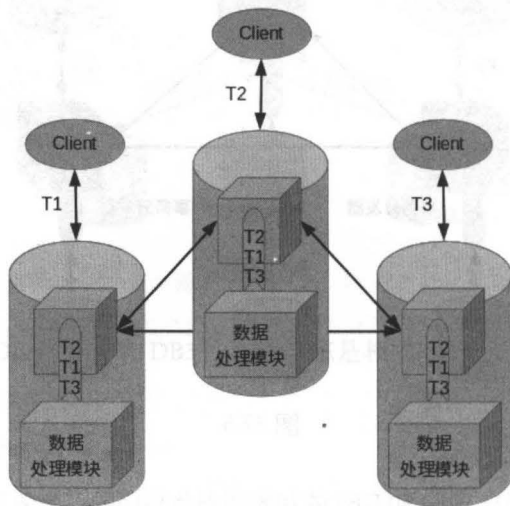


图 27.6

用户通过数据库服务器的用户接口执行事务。数据库服务器收到事务请求后, 首先交由事务分发模块处理。事务分发模块将事务汇总排序, 然后依次交由数据处理模块去执行这些

事务。如果去掉内部的细节,就会发现这是一个非常简洁的数据库集群方案。通过 Group Replication 构建的 MySQL 集群就是这样一个分布式的高可用 MySQL 系统。整个集群的架构如图 27.7 所示。

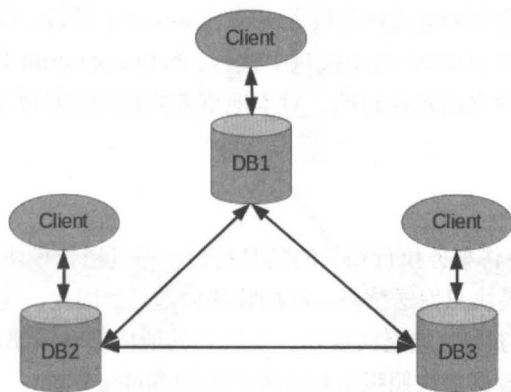


图 27.7

深入理解 Group Replication 中事务的执行过程

从上一节中的模型可以知道, Group Replication 插件中最重要的功能就是事务分发器的功能。Group Replication 中实现的事务分发器和模型中的略有不同。在模型中,假设事务分发器分发的是事务的 SQL 语句,也就是说事务分发器的处理是在事务执行前。而在 Group Replication 中,事务分发器分发的是 Binlog Event,事务分发器的处理是在事务执行即将结束的时候。Group Replication 将这称作乐观的事务执行策略,可以带来更好的性能。但在这种策略下,多个成员上的事务可能发生冲突。Group Replication 需要一个冲突检测机制来发现并处理冲突。由于 Binlog Event 的执行和 SQL 语句的执行过程是不一样的, Group Replication 就用到异步复制中 Binlog Event 的执行模块。为了更好地理解 Group Replication 事务的执行过程,这里将 Group Replication 处理的事务分为本地事务和异地事务。本地事务指的是用户 Session 中或异步复制线程中执行的事务。异地事务指的是 Group Replication 中传播的由 Binlog Event 构成的事务。

根据事务处理过程中的不同处理步骤, Group Replication 中事务分发器的功能划分为以下四部分。

- 本地事务控制模块
- 成员间的通信模块
- 全局事务认证模块
- 异地事务执行模块

本地事务控制模块

MySQL 通过 API 向插件提供了事务执行过程中几个重要阶段的监控接口, Group Replication 通过这些接口来监控和控制事务的执行。MySQL 的事务在提交时, 内部会分成三个阶段: 准备 (prepare) 阶段、记录 Binlog 文件阶段和提交 (commit) 阶段。Group Replication 对本地事务的控制逻辑在 `before_commit` 这个接口中执行。`before_commit` 是在事务的 prepare 阶段之后, 写 Binlog 文件阶段之前被执行的。对本地事务的控制包括以下三个步骤。

发送事务信息

Group Replication 首先会将事务执行相关的信息打包, 通过通信模块的接口发送给本地的通信模块。本地事务控制模块只发送信息, 不接收任何信息。因此, 这个通信是异步过程。只要本地的通信模块接收了消息就返回成功。发送到其他成员是成员间通信模块的职责。事务信息包括主键信息、数据库快照版本和事务产生的 Binlog Event。

- 主键信息是 Server 层生成 Binlog Event 的时候一同生成的。前面章节说过, 是否记录主键信息是通过 `transaction_write_set_extraction` 变量来控制的。主键信息中记录的并不是主键字段的值, 而是字段值加上库名、表名等哈希值。
- 数据库快照版本是当前 MySQL 的全局变量 `gtid_executed` 的值。它包含了当前事务提交时所有已经执行了的事务的 GTID, 代表了当前事务执行时数据库的状态, 因此称为数据库的快照版本。
- 当发送事务信息时, Binlog Event 还没有写入 Binlog 文件。因此, Binlog Event 是从当前线程的 Binlog Cache 中获取的, 而不依赖于 Binlog 文件。
- `Transaction_context_log_event`, 本地成员的 UUID、主键信息和数据库快照版本会被封装进 `Transaction_context_log_event` 中, 和事务产生的 Binlog Event 一起发送出去。`Transaction_context_log_event` 放在其他 Binlog Event 的前面。

等待全局事务认证模块的认证结果

在事务信息发送成功后, 事务会被阻塞, 开始等待全局事务认证模块的认证结果。事务认证完成后, 全局事务认证模块会唤醒当前事务的线程, 让事务继续执行。

认证结果的处理

这个过程实际上是由 MySQL 的代码执行的, 而不是由 Group Replication 的代码执行的。在哪里执行的并不重要, 只要理解这个过程就好。事务继续执行后, 会检测认证结果。如果认证成功, 就继续提交事务; 如果认证失败了, 就会返回错误。然后由 MySQL 来执行 Rollback 的逻辑。

成员间的通信模块

通信模块分为三部分，如图 27.8 所示。

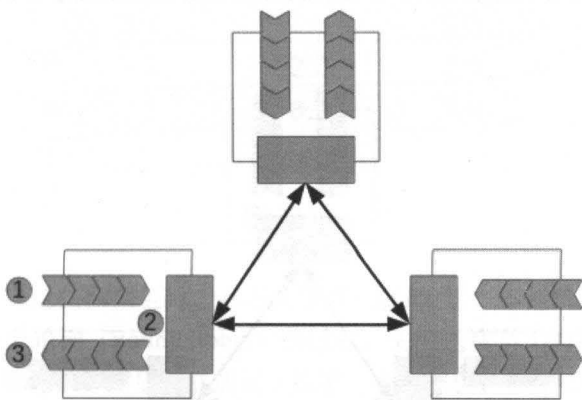


图 27.8

1. 本地数据接收部分负责接收本地成员上其他模块的数据发送请求，接收到的数据包被放入本地数据队列等待处理。
2. 成员间的通信部分负责和其他成员通信。通信工作包括：从本地数据队列读取数据包发送给其他成员，以及接收其他成员发送过来的数据包。各个成员之间的通信使用了 Paxos 协议，Paxos 是一个分布式的一致性协议。
3. 全局数据包发送部分将所有的数据包按顺序返回给本地成员上的全局事务认证模块。

当各个成员的通信模块接收到上层模块的数据发送请求时，这些并发的数据请求是无序的，如图 27.9 所示。

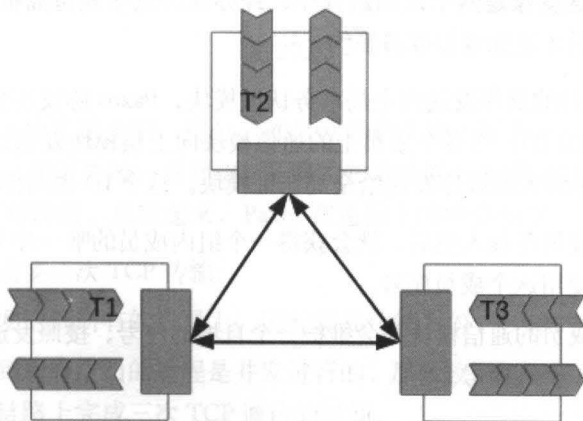


图 27.9

3 个成员分别有 1 个数据包, 分别是 T1、T2 和 T3 产生的数据包。这些并发、无序的数据包会通过 Paxos 协议汇聚到每个成员上, 并且排序。最终每个成员的通信模块都会拥有同样的数据包, 这些数据包会按照同样的顺序发送到各自成员上的全局事务认证模块, 如图 27.10 所示。

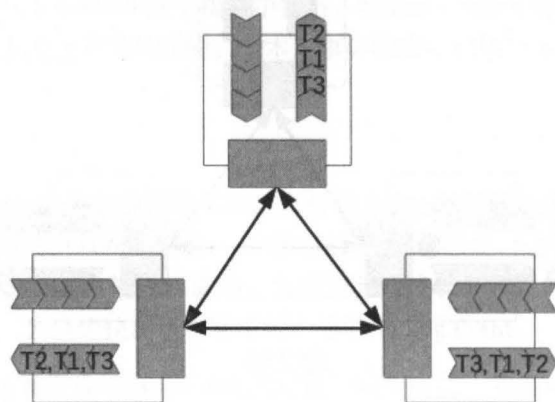


图 27.10

Paxos 协议的核心工作就是对所有的数据包进行汇聚和排序。为了完成上面的功能, Paxos 协议本身会进行三次 TCP 通信, 如下。

- 发送数据包给其他成员的通信模块。
- 其他成员的通信模块回收收到数据包。
- 当超过半数的通信模块 (包括它自己) 回应后, 发送消息告诉所有节点, 这个数据包同步成功。只有当 Paxos 协议的三个步骤成功完成后, 通讯模块才会把这个数据包发送给全局认证模块。这就像是两个公司谈合作, 经办人经过多轮协商把所有的条款都谈妥, 没有异议了, 然后才通知老板准备签约。

为了保证数据按照同样的顺序发送到全局事务认证模块, Paxos 协议还会为每个同步成功的数据包分配一个唯一的 ID, 当各个成员上的通信模块向上层模块发送这些数据包时, 会按照数据包的 ID 以从小到大的顺序发送给全局认证模块。这个 ID 由两部分组成, 分别如下。

- 成员序号: 每个成员在加入组后, 就会获得一个组内成员的唯一序号, 这个成员发出的所有数据包都会使用这个成员序号。
- 自增序号: 每个成员的通信模块都会维护一个自增的序号, 按照发送数据包的顺序, 给每个数据包分配一个顺序号。

如图 27.11 所示, 假设一个组有三个成员, 那么顺序号为 1 的成员上的数据包 ID 是 1:1, 2:1, ..., N:1。顺序号为 2 的成员上的数据包 ID 是 1:2, 2:2, ..., N:2。顺序号为 3 的成员 3 上的数据包

的 ID 是 1:3,2:3,...,N:3。这些数据包包排序后的顺序就是 1:1,1:2,1:3,2:1,2:2,2:3,...,N:1,N:2,N:3。

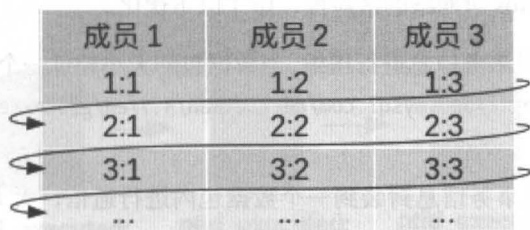


图 27.11

当向全局认证模块发送这些数据包时，必须按照 ID 连续发送，不能跳过某个 ID 的数据包。本质上来说，这是一个轮询 (Round Robin) 的过程。先发送成员 1 的第一个数据包，接着发送成员 2 的第一个数据包，然后发送成员 3 的第一个数据包，如图 27.12 所示。

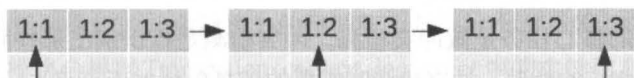


图 27.12

如果在这个过程中，成员 2 上的数据包还没有同步成功呢？这时候就得等着，直到成员 2 的第一个数据包同步成功后，才能返回给全局认证模块，然后继续返回成员 3 的数据包。假如成员 2 是空闲的，且本地没有任何事务处理呢？这种情况下，成员 2 需要发一个空操作指令 (NOP)，告诉其他成员跳过 ID1:2 的数据包。这个指令是当成员 2 收到 ID1:1 的数据包同步成功的消息 (Paxos 的第三个 TCP 消息) 后发出的，如图 27.13 所示。

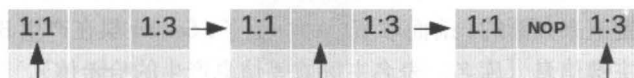


图 27.13

以上介绍的只是 Paxos 协议很少的一部分，如果读者想详细了解 Paxos 协议，还是要查阅更多的资料才行。本章之所以要做一点介绍，是因为知道这部分对理解 Group Replication 在不同网络中的性能是有帮助的。总结起来，Paxos 在通信上的特点如下。

- 数据包同步成功需要三次 TCP 传输。
- 每个数据包都要发送到所有的成员上，因此需要传输多份，传输的数据量会被放大。
- 假设数据包发送到所有成员的过程是并发进行的，那么数据包同步成功需要的时间是成员间最慢的那条链路上完成三次 TCP 通信的时间。

这些特点决定了 Group Replication 在延迟大、带宽窄的网络中的效率会是比较低的。Group Replication 为了提高 Paxos 对网络的适应性,做了以下优化。

- 使用 LZ4 压缩算法对事务信息进行压缩,当数据包的大小超过一个阈值时会被自动压缩。具体信息参见: <https://dev.mysql.com/doc/refman/5.7/en/group-replication-message-compression.html>。
- Paxos 会将多个本地事务信息封装到一个数据包内进行通信,大大地减少了 Paxos 通信的次数。

全局事务认证模块

全局事务认证模块有一个消息队列,用来存放所有收到的消息。这些消息主要是事务的 Binlog Event,也有一部分状态和控制消息。状态表 `replication_group_member_stats` 中的字段 `COUNT_TRANSACTION_IN_QUEUE` 指的就是这个队列中的事务数量。

全局事务认证模块的核心任务是做冲突检测。冲突检测是指识别出那些同时修改了同样数据的事务,并做出相应的处理。冲突检测时需要的信息包括如下三点。

- 主键信息。
- 事务执行时的数据库快照版本。
- 执行事务的 MySQL 服务器的 UUID。

冲突检测需要的信息

Group Replication 的冲突检测中以数据行为单位,两个事务是不是修改了同样的数据,是通过事务所修改的主键值来判断的。前面章节已经介绍了,Server 层在产生 Binlog Event 时,会同时记录所修改的主键信息(库名、表名主键值等信息产生的哈希值)。当发现两个事务修改了同样的数据后,如何来判断这两个事务是不是同时执行的呢?这里用到了数据库快照版本。数据库快照是数据库的一个瞬时状态,每个写操作都会导致数据库的状态变化,不同的状态用不同的快照版本来表示。快照版本是用 GTID 来表示的,每个写事务都会产生一个唯一的 GTID,这个 GTID 由全局事务认证模块产生,且在事务提交时会被添加到全局变量 `gtid_executed` 中。因此, `gtid_executed` 的内容就是 MySQL 数据库的快照版本。

图 27.14 是数据库快照版本变化的示例图。图中,事务 T1 开始执行时所基于的数据库快照版本为空,T1 提交后数据库的版本变为“group_name:1”,group_name 是 Group Replication 组的 UUID。事务 T2 的执行则基于“group_name:1”的快照版本,这个数据库的快照版本、主键信息及其 MySQL 服务器的 UUID 一起被封装到 `Transaction_context_log_event` 上,和其他事务产生的 Binlog Event 一起发送到全局事务检测模块。

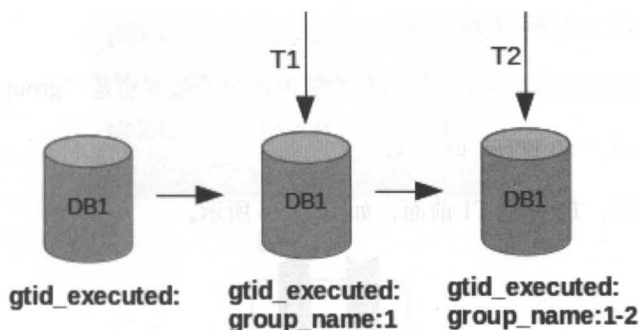


图 27.14

冲突检测数据库

全局事务认证模块中还维护了一个冲突检测数据库，它是主键哈希 + 快照版本的列表。快照版本存储的是最后一个修改此主键事务的快照版本加上这个事务的 GTID。收到事务信息后，全局事务认证模块会根据 `Transaction_context_log_event` 中的主键信息，从冲突检测数据库中检索出所有主键的快照版本和该事务的快照版本进行比对。当前事务的快照版本必须包含检索出的所有主键快照版本中的 GTID，否则就是有冲突。

理解冲突检测的过程

下面通过一个例子来理解冲突检测的原理，如图 27.15 所示。

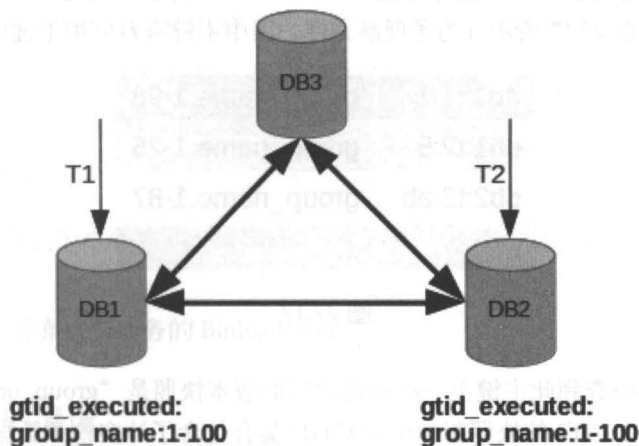


图 27.15

假设当前所有成员的数据库状态是一致的，它们的 `gtid_executed` 值都为 “`group_name:1-100`”。这时，DB1 上执行了事务 T1，T1 看到的数据库快照版本就是 “`group_name:1-100`”。

`UPDATE t1 SET c2 = 5 WHERE pk = 1;`

与此同时, DB2 上执行了事务 T2, T2 看到的数据库快照版本也是 “group_name:1-100”。

`UPDATE t1 SET c2 = 10 WHERE pk = 1;`

经过通信模块排序后, T2 排在 T1 前面, 如图 27.16 所示。

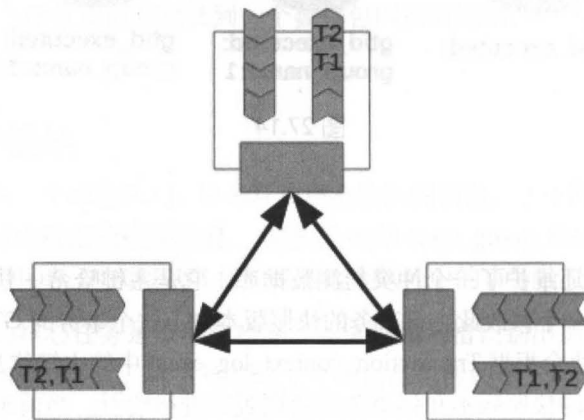


图 27.16

因此, 在所有成员全局冲突检测模块中都是先处理 T2, 再处理 T1。在处理 T2 时, 首先根据 `Transaction_context_log_event` 中的主键信息从冲突检测数据库中查找该主键上一次被修改后的快照版本。有可能查到, 也有可能查不到。查不到就意味着没有冲突。假设冲突检测数据库中的内容如图 27.17 所示 (为了理解方便, 图中主键哈希值用主键值代替)。

db1:t1:1	group_name:1-98
db1:t2:5	group_name:1-25
db2:t2:ab	group_name:1-87
...	...

图 27.17

从冲突检测数据库中查到此主键上一次被修改后的版本快照是 “group_name:1-98”。和 T2 的快照版本比较得知, T2 的快照版本中的 GTID 集合包含了冲突检测数据中查找到的快照版本的 GTID 集合。也就意味着, T2 在 B 成员上执行时, 上一次该主键的更新已经在 B 成员上执行了, 所以 T2 和上一次该主键的更新之间是不冲突的。接着, 全局事务检测模块会为 T2 分配一个 GTID, 并更新冲突检测数据库中此主键的快照版本。假设 T2 分配的 GTID 是 “group_name:101”, 更新后的冲突检测数据库如图 27.18 所示。

db1:t1:1	group_name:1-101
db1:t1:2	group_name:1-25
db2:t2:ab	group_name:1-87
...	...

图 27.18



注意：这个快照版本是指该主键被 T2 更新后的快照版本，因此也包含 T2 自己的 GTID。所以，冲突检测数据库中该主键的快照版本是“group_name:1-101”。

冲突检测完成后，还有后续的处理，稍后再讲。T2 处理完后，开始下一个事务 T1 的冲突检测。首先，也是从冲突检测数据库中查找到该主键的快照版本，查到的结果是“group_name:1-101”。和 T1 的快照版本对比发现，T1 的快照版本不包含“group_name:101”，这就可以判定 T1 和之前一次的主键修改（T2 的修改）是同时进行的，是冲突的。

冲突处理

冲突检测完成后，全局认证模块接下来的处理是有本地事务和异地事务区分的。Transaction_context_log_event 中记录了产生这个事务的 MySQL 服务器的 UUID。根据这个 UUID，就能判断出这是一个本地事务还是异地事务。对于本地事务的处理如下。

- 如果没有冲突，唤醒这个事务的线程，并且告诉它完成提交操作。
- 如果有冲突，唤醒这个事务的线程，并且告诉它发生冲突，需要回滚。
- 不论是否有冲突，Binlog Event 都会被丢弃。

对于异地事务的处理如下。

- 如果没有冲突，将这个事务的 Binlog Event 写入 Relay Log 中，让 group_replication_applier 通道去执行。
- 如果有冲突，则丢弃这个事务的 Binlog Event。

冲突检测数据库的清理

随着使用时间越来越长，冲突检测数据库中维护的主键信息会越来越多，这些主键信息将占用巨大的内存。为了减少内存的使用并提高查询效率，全局事务认证模块需要定期的清理冲突检测数据库。如果一个事务已经在所有的成员上执行了，其他事务的执行肯定不会和它有冲突，因此这个事务的所有主键信息就可以从冲突检测数据库中移除。主键信息的

清理是依据各个成员上的全局变量 `gtid_executed` 中的 GTID 集合来做的。全局认证模块启动了一个广播线程，每 60 秒将自己的 `gtid_executed` 中的 GTID 集合广播到所有的成员上。全局事务认证模块收到所有成员的 GTID 集合后，取它们的交集。这个交集中包含的就是那些已经在所有成员上执行了的事务 GTID 的集合，称作全局完成的 GTID 集合。全局事务认证模块会将冲突检测数据库所有主键的快照版本和全局完成的 GTID 集合进行比对，如果快照版本中的 GTID 集合是全局完成的 GTID 集合的子集，则这个主键的信息就会从冲突检测数据库中清除掉。

`replication_group_member_stats` 表中的 `TRANSACTION_COMMITTED_ALL_MEMBERS` 显示的就是全局完成的 GTID 集合。


计算基于主键的逻辑时钟

在执行 Binlog Event 时，Group Replication 采用了基于主键的并发机制。在这种机制中通过主键来判断是否修改了相同的数据，各种情况如下。

- 修改了不同数据的事务会安排并发执行。
- 修改了相同数据的事务会安排顺序执行。
- DDL 不能和任何事务并发执行，必须等待它前面的所有事务执行完毕后才能开始执行。后面的事务也必须等待 DDL 执行完毕后，才能开始执行。

基于主键的并发机制，刚好可以通过逻辑时钟的方式来表达。因此 Group Replication 重用了多线程复制中 `Logical_clock` 并发复制的功能。基于主键的并发实现起来很简单，只需要根据主键信息计算出事务的逻辑时间，并更新 `Gtid_log_event` 中的 `last_committed` 和 `sequence_number` 的内容即可。`group_replication_applier` 在执行这些事务时，按照 `Logical_clock` 方式进行并发就可以了，不需要任何改动。由于需要用到主键信息，因此计算过程是在冲突检测的过程中完成的。全局事务认证模块维护了一个全局的事务顺序号，它会给每个认证成功的事务分配一个顺序号。这个顺序号就是事务的 `sequence_number`，会存储到冲突检测数据库中。当前事务的 `last_committed` 的计算和存储在冲突检测数据库中的 `sequence_number` 有关。首先，从冲突检测数据库找出该事务修改的所有主键的 `sequence_number`，即上一次被修改时的 `sequence_number`，取其中的最大值来作为当前事务的 `last_committed`。

假设正在处理的事务如下。

```
 UPDATE t1 SET c2 = 5 WHERE pk=1 OR pk = 2;
```

冲突检测数据库中的顺序如图 27.19 所示。

那么该事务的 `last_committed` 是 120（120 和 33 的最大值）。如果查不到上次记录，则将 `last_committed` 设置为上一次 DDL 的 `sequence_number`，这保证 DDL 后面的事务不会和 DDL

并发执行。如果当前事务是 DDL，则 `last_committed` 会设置成它自己的 `sequence_number-1`，从而保证 DDL 前面的所有事务都被执行完后它才会被执行。

db1:t1:1	group_name:1-101	120
db1:t1:2	group_name:1-25	33
db2:t2:ab	group_name:1-87	90
...	...	

图 27.19

异地事务执行模块

为了执行异地事务的 Binlog Event，Group Replication 会自动创建一个名为 `group_replication_applier` 的通道。这个通道的 Receiver 线程是关闭的，不会从其他成员上去复制 Binlog Event。所有的 Binlog Event 都是由全局事务认证模块通过 API 写入 Replay Log 的。Binlog Event 的执行过程和异步复制没有区别，但是在执行过程中 `group_replication_applier` 所执行的事务使用了特权行锁。在对数据加行锁的时候，如果 `group_replication_applier` 发现有本地事务已经对数据加了行锁，那么 `group_replication_applier` 不会等待本地事务执行完毕，而是立刻将本地事务回滚掉，然后继续执行；而本地事务的 Session 则会返回错误。这其实很好理解，本地事务和 `group_replication_applier` 更新了同样的数据，而且是同时执行的，因此才会返回错误。即便本地事务到了全局事务认证模块，也会因为检测出冲突而被回滚掉。相比而言，由前者回滚本地事务效率更高。

事务流程的总结

事务在 Group Replication 中的执行过程可以总结为以下三个部分。

- 网络传输。
- 事务在本地的执行过程。
- 事务在异地的执行过程。

网络传输

Group Replication 通过 Paxos 协议来传播事务信息。Paxos 保证所有的事务信息按照同样的顺序传播到所有的成员上，如图 27.20 所示。

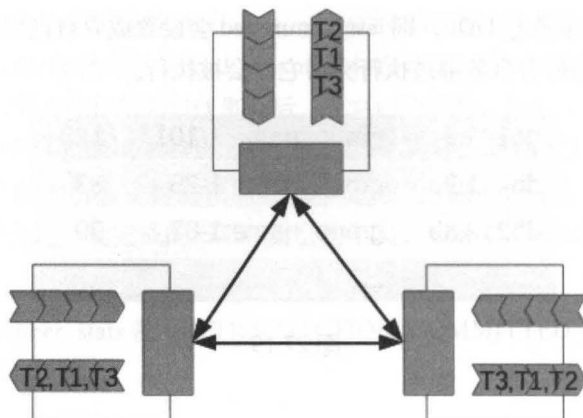


图 27.20

事务在本地成员上的执行过程

事务在本地提交时（prepare 之后，写 Binlog 之前），将事务信息发送至通信模块，然后开始等待事务认证结果。通信模块将事务排序后发送到本地成员的全局事务认证模块。全局事务认证模块做完冲突检测后，唤醒该事务继续执行（如果认证成功）或回滚（如果认证失败），如图 27.21 所示。

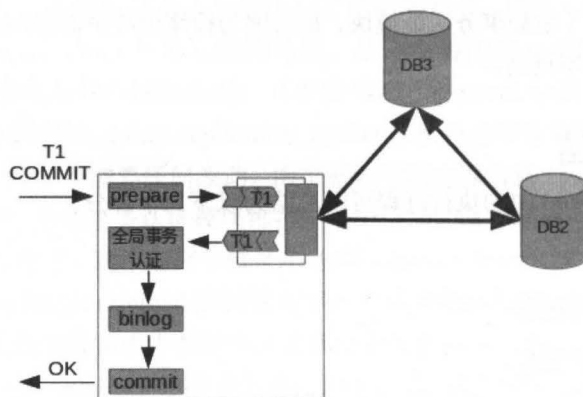


图 27.21

事务在异地的执行过程

通讯模块将事务排序后发到全局事务认证模块。全局事务认证模块认证成功后，将该事务的 Binlog Event 写入 Relay Log，由 group_replication_applier 通道来执行。如果全局事务认证

模块认证失败，则会丢弃该事务的 Binlog Event，如图 27.22 所示。

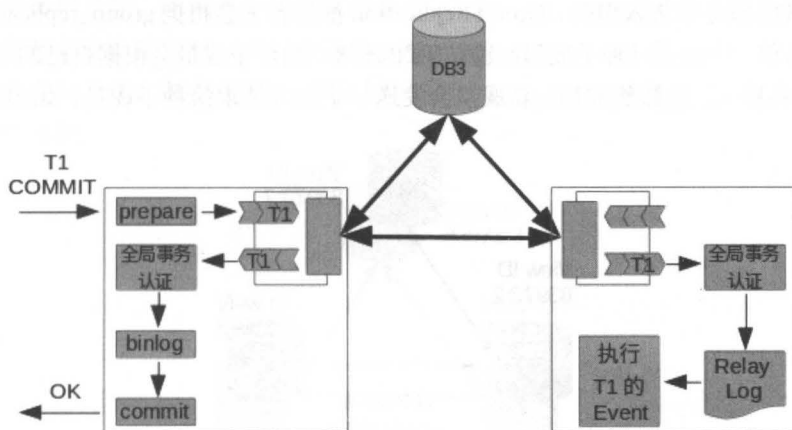


图 27.22

深入理解成员加入组的过程

Group Replication 的主要逻辑可以划分为两部分：事务的执行逻辑和成员的管理逻辑。前面已经介绍了事务的执行逻辑，本节介绍成员的管理逻辑。

组视图

成员管理中有一个很重要的概念叫作组视图 (Group View)，或者简称为视图 (View)。视图是指 Group Replication 组在一段时间内的成员状态。在这个时间段内没有成员变化，即没有成员加入也没有成员退出。如果成员发生了变化，成员状态就变化了，于是它就进入了另外一个视图。不同的视图之间通过视图 ID (View ID) 来进行区分。视图随时间的变化有先后顺序，因此 View ID 也是有先后顺序的。View ID 被定义为两个部分，分别如下。

- 前缀部分 (固定部分)：前缀部分是一个随机数。这个值在组初始化时产生。之后，所有 View 的前缀部分都使用这个随机数，因此也被称为固定部分。
- 序号部分：序号部分是数值。初始化时，第一个视图的序号从 1 开始，以后每次视图变化序号递增。

用户可以通过 `replication_group_member_stats` 表查看当前的 View ID。View ID 显示格式如下。

14870305055874300:2

加入组时视图的切换

当一个 MySQL 服务器加入组时，Group Replication 插件首先会根据 `group_replication_group_seeds` 的内容和一个成员（种子成员）建立 TCP 连接。而种子成员会根据自己的 IP 白名单检查是否允许其接入。连接建立后，新成员会发送一个加入请求给种子成员，如图 27.23 所示。

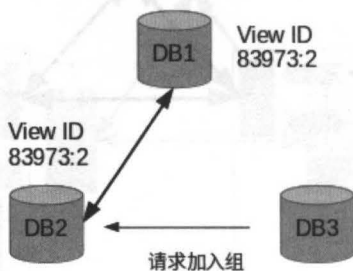


图 27.23

收到请求后，种子成员广播视图变化的消息给所有成员（包括申请加入的成员），如图 27.24 所示。

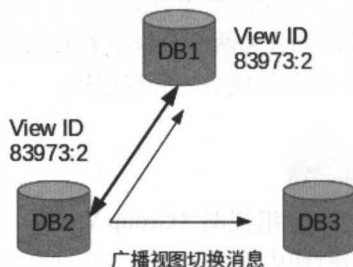


图 27.24

各个成员收到消息后开始做视图切换。首先，每个成员都会广播一个状态交换消息出去，如图 27.25 所示。

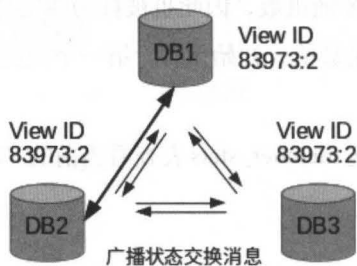


图 27.25

接着，各个成员开始接收状态交换消息。状态交换消息中包含了成员的当前状态和信息。成员收到状态交换信息后，将消息中的成员信息更新到自己的成员列表中。当收到所有成员中的最后一个状态交换消息时，通信模块将完整的新视图以视图数据包的形式返回给全局事务认证模块进行处理。整个视图的切换过程至此结束，视图切换的整个过程不影响在线成员对外提供服务。

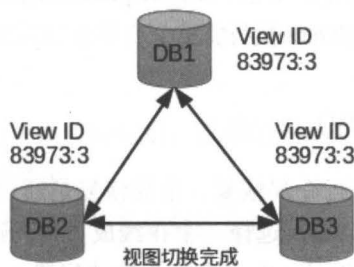


图 27.26

离开组时的视图切换和加入组时的视图切换过程基本一样，这里就不做介绍了。

View_change_log_event

状态交换消息和事务数据包一样都是通过 Paxos 协议发送的，因此它们之间也是有序的。事务数据包以视图数据包为分界线，划分到不同的视图组中。它前面的事务数据包属于前一个视图组的事务，而后面的数据包都是属于当前视图组的事务。全局事务认证模块会对每一个视图数据包创建一个 View_change_log_event，这个 Event 会被写入 Relay Log，然后被执行，最终会出现在 Binlog 中，因此 Binlog 里的 Event 也被 View_change_log_event 划分到不同的视图组中。View_change_log_event 里面记录了当前的 View ID，还有冲突检测数据库的信息。当记录 View_change_log_event 到 Binlog 中时，它会被封装到一个事务中，并且有自己的 GTID，包括以下几个 Event。

```
Gtid_log_event;
Query_log_event("BEGIN");
View_change_log_event;
Query_log_event("COMMIT");
```

恢复

视图切换完成后，成员就正式加入到了组内。它可以收到当前视图任何事务的数据包，但是视图切换前的数据包是收不到的。所以，在 MySQL 服务器加入组后，不能立即对外提供服务，而需要执行一些操作将自己缺失的数据从其他成员上复制过来。这个过程叫作恢复 (Recovery)。一个成员加入组后，会立即将自己的状态设置为 RECOVERING，开始执行恢复

的过程。恢复过程分为本地恢复、全局恢复和缓存事务执行三个步骤。

本地恢复 (Local Recovery)

如果这个成员曾经加入过这个组, 它的 `group_replication_applier` 通道的 Relay Log 中可能还有一些 Event 没有被执行到数据库中。因此, Group Replication 插件首先会启动 `group_replication_applier` 通道, 将本地的 Binlog Event 执行完毕。

全局恢复 (Global Recovery)

本地恢复执行完毕后, 开始进行全局恢复。全局恢复是通过 `group_replication_recovery` 进行的。Group Replication 插件会随机选择一个在线成员作为这个通道的 Master, 这个被选择的成员叫作 Donor。Group Replication 插件会自动配置 `group_replication_recovery` 通道, 并且启动这个通道进行复制, 复制时使用的是 GTID 复制。全局恢复有容错能力, 如果 `group_replication_recovery` 通道的接收线程 (Receiver) 无法连接到当前的 Donor 或当前 Donor 上的 Binlog 已经删除, 则会选择其他成员作为 Donor 进行复制。在启动 `group_replication_recovery` 通道时, Group Replication 插件会告诉它: 当碰到 View ID 等于当前 View ID 的 `View_change_log_event` 时停止。当执行完到这个 `View_change_log_event` 后, `group_replication_recovery` 通道会将这个 Event 中的冲突检测数据库初始化到 Group Replication 插件中, 然后停止运行, 如图 27.27 所示。

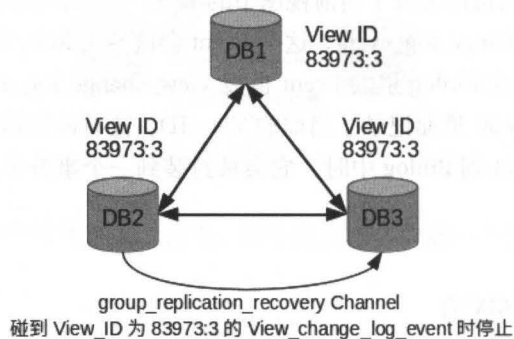


图 27.27

缓存事务执行

在执行全局恢复过程的同时, 全局事务认证模块还会收到当前视图中产生的事务数据包。这些数据包会被缓存起来直到全局恢复执行完毕后, 全局事务认证模块才开始处理这些事务。当缓存的事务全部执行完毕后, 该成员才能设置为 ONLINE 状态。当然用户也可以让成员在

缓存的事务执行之前就上线，Group Replication 提供了参数 `group_replication_recovery_complete_at` 来控制上线时间。上线不仅仅是状态的改变，在多主模式时，上线意味着只读模式会被关闭，上线后成员就能够接收用户的写操作了。

手动恢复

虽然 Group Replication 的恢复过程是很自动化的，但是并不完美，在有些情况下还是需要手动进行恢复的。

- 当 Group Replication 运行了很长时间后，以前的 Binlog 可能已经被删除了，这时就无法自动通过全局恢复来复制数据。
- 当要复制的 Binlog 很大时，使用异步复制的全局恢复效率比较低，不能在短时间内完成。

当要加入组内的 MySQL 服务器符合以上情况时，可以先手动将数据库恢复到一个比较接近的时间点，然后再加入 Group Replication 组。

28

MySQL Document Store 面面观

本章作者为特邀撰稿人：杜修文

MySQL 面世 20 多年来，在大家的认知中始终是一个关系型数据库。但是从 2016 年起 MySQL 有了一个革命性的变化——它除了是一个说 SQL 语言的关系型数据库，也学会了第二种语言——JSON。现在，MySQL 也是一个 NoSQL 文件型数据库。

大约在两年前，Oracle 的 MySQL 产品部门认识到软件工程正朝向锥形法、快速开发和开发运维（DevOp）的方向发展。传统的关系型数据库虽以严谨著称，强调数据完整性和通过正规化带来查询弹性，但相对地，要以较长时间变动数据架构的做法支持现今软件工程的方法论，存在一定程度上的适用性问题。同时，过度强调正规化也会因常常需要通过多表的连接（JOIN）才能查询一份完整的数据而存在性能不佳的问题。所以，MySQL 决定启动一个代码为 MySQLng（MySQL Next Generation）的项目。这个工程投入了包括数据库核心、前端工具和链接器（Connector）部门在内的大量开发人力，成就了 MySQL 在 2016 年推出的以 MySQL 当作文件储存（MySQL as a Document Store）的新功能。

MySQL 的架构也因此出现了新的变化，MySQL 在 5.7.12 版本以后就是一个完全支持 JSON 文档的文件型数据库。它从数据库核心到前端工具及支持各种前端应用技术的链接器（Connector），都出现了许多新功能，可分为如下四个部分，可结合图 28.1 所示来看。

- 数据库核心（即图中 28.1 中 Core 部分）增加了新的 JSON 数据类型和 JSON 函数。
- 新加入了一个名为 MySQL X 的插件（对应图 28.1 中 X Protocol Plugin 部分），默认使用 33060 端口，使数据库能通过 X Protocol 和 X DevAPI 运行应用程序发送的 CRUD 操作。

- 新的 MySQL 客户端——MySQL Shell (如图 28.1), 以 JSON 文档的 CRUD 方法直接操作数据。
- 扩增多种链接器 (如图 28.1 中所示的 X DevAPI, 包括 Connector/J、Connector/Python、Connector/Nodejs、Connector/C、C++, 以及 Connector/PHP) 的功能使它们可通过包裹 (wrap) X DevAPI, 而使用 X Protocol 以纯 NoSQL 的方式对数据库的 JSON 文档进行操作, 使整个应用的开发完全没有 SQL 的影子。

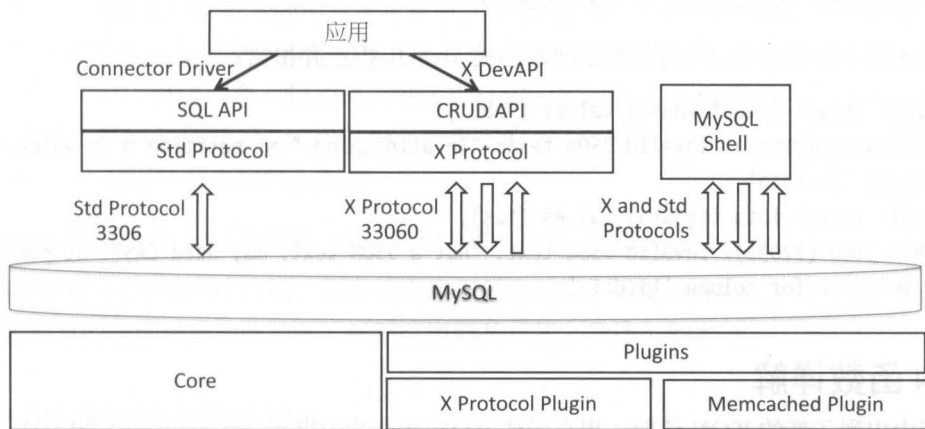


图 28.1

以下对这四个支持以 MySQL 当作文件存储 (MySQL as a Document Store) 的各个模块特性进行详细说明。

新的 JSON 数据类型和 JSON 函数

MySQL 数据库内核中加入了新的 JSON 数据类型和 JSON 函数, 下面来逐一详细说明。

JSON 数据类型

自 MySQL 5.7.8 起, 这项功能就包含在 MySQL 中了。可以用熟悉的 MySQL 客户端程序登录数据库, 建一个有 JSON 字段的表, 再将 JSON 文档插入这个字段, 具体的范例如下所示。

JSON 字段以 binary 为基础, 再于字段头标示文件中各对象的位移 (offset), 如此能让 MySQL 快速地找到查询所指定的对象。

创建一个表, 里面有个 data 字段, 注意 data 字段是 JSON 数据类型, 不是 text 或 varchar 数据类型:

```
mysql> CREATE TABLE t1 (data JSON);
Query OK, 0 rows affected (0.08 sec)
```

以字符串形式将数据插入表中, 对象名称要在引号内:

```
mysql> INSERT INTO t1(data) VALUES ('{ "series": 1}'), ('{ "series": 7}'),
    ('{ "series": 3}'), ('{ "series": 4}'), ('{ "series": 10}'), ('{ "series": 2}'),
    ('{ "series": 6}'), ('{ "series": 5}'), ('{ "series": 8}'), ('{ "series": 11}');
Query OK, 10 rows affected (0.02 sec)
Records: 10 Duplicates: 0 Warnings: 0
```

如果插入的不是符合 JSON 规范的数据则会被 MySQL 拒绝并报错:

```
mysql> insert into t1(data) values ('abc');
ERROR 3140 (22032): Invalid JSON text: "Invalid value." at position 0 in value for
column 't1.data'.
mysql> insert into t1(data) values (123);
ERROR 3140 (22032): Invalid JSON text: "not a JSON text, may need CAST" at position
0 in value for column 't1.data'.
```

JSON 函数详解

数据库中出现了新的 JSON 函数, 用来操作 JSON 文档的内容, 例如 JSON_EXTRACT。

```
mysql> SELECT * FROM t1 WHERE json_extract(data,"$.series") >= 7 AND json_extract(
data,"$.series") <=10;
+-----+
| data |
+-----+
| {"series": 7} |
| {"series": 10} |
| {"series": 8} |
+-----+
3 rows in set (0.00 sec)
```

目前有 23 个 JSON 函数 (以后随着更新的版本推出, 可能会增加更多的 JSON 函数以完善处理 JSON 文档的功能), 根据这些函数使用的时机可分为检索 JSON 文档信息、更改 (由路径参数指定的) JSON 文档的内容、建立 JSON 文档、取出 (由路径参数指定) 部分 JSON 文档内容和辅助性函数等 5 类。

由于 JSON 函数会以路径 (path) 指定到 JSON 文档中一定的位置, 取出或参照部分文档的信息, 所以要在介绍 JSON 函数前, 先说明路径语法, 如下。

- 在 SQL 命令中, 要先找到存放 JSON 文档的 JSON 字段, 其语法为 [[database].table.]field, 找到整个 JSON 文档。

- 找到字段后，再根据路径指向存于字段的 JSON 文档的某一部分。在 JSON 文档中路径的语法如下。
 - `$`——JSON 文档的根。
 - `.identifier`——指定名称到该名称所对应的值或对象，例如 `$.user`，找出 JSON 文档中的 `user` 对象或值。
 - `[array-position]`——指定数组中某一位置的内容（由 0 开始），例如 `$.user.address[1]`，找到 JSON 文档 `user` 对象之下 `address` 数组第二个位置的地址对象。
 - `*`——指定某一对象下的所有成员对象，例如 `$user.*` 代表找出 JSON 文档中 `user` 对象下所有的成员对象，可能包括名字、生日及所有的地址等。
 - `**`——可以当成一个 wild card，用法是 `[prefix]**suffix`，`prefix` 是选项，`**` 不会作为结尾，例如 `$user**.phone` 指向 `user` 对象下的 `phone` 对象。
 - `[*]`——找出数组中的所有内容。

JSON path 的范例将在下一段 `JSON_EXTRACT` 函数中再做详细的说明。

接下来再介绍 5 大类 JSON 函数，其中最常用到的函数是 `JSON_EXTRACT` 或 `->`，以及 `->>`。其他函数列出的范例稍微琐碎了些，仅用以说明其功能，供读者们参考。


检索 JSON 文档的内容或属性

这类函数包括如下几项。

- `JSON_VALID(val)`，检查传入的数据是否是一个规范的 JSON 文档。目前是以较旧的 RFC2627 为标准，合规传出 1，不合规传出 0，若传入 NULL 值，则传出的也是 NULL 值。这个函数能让我们在应用中就先测试所拿到的数据是否是一个 JSON 文档，如果不是，则应用开发者在应用中就能先处理。
- `JSON_TYPE(json_val[,path])`，检索指定的 JSON 文档或文档中以路径 `path` 指向的数据类型。这个函数的第二个参数 `path` 之后有多个路径。
- `JSON_KEYS(json_doc[,path])`，传回指定 JSON 文档的最上一层（或是指定路径）的键值 (`identifier`)，下面举例说明其用途。




```
# 文件的根下有两个 identifier : "a" 和 "b"
mysql> SELECT JSON_KEYS('{ "a": 1, "b": { "c": 30 } }');
+-----+
| JSON_KEYS('{ "a": 1, "b": { "c": 30 } }') |
+-----+
| ["a", "b"]                                |
+-----+
```

 # 文件“根”的成员“b”对象下有一个identifier: “b”

```
mysql> SELECT JSON_KEYS('{ "a": 1, "b": { "c": 30 } }', '$.b');
+-----+
| JSON_KEYS('{ "a": 1, "b": { "c": 30 } }', '$.b') |
+-----+
| ["c"] |
+-----+
```

- `JSON_LENGTH(json_doc, path)`, 传回指定 JSON 文档最上一层（或是指定路径）的成员数，如下。

 # 传入 `JSON_LENGTH` 函数的参数到有三个元素的数组

```
mysql> SELECT JSON_LENGTH('[1, 2, { "a": 3 } ]');
+-----+
| JSON_LENGTH('[1, 2, { "a": 3 } ]') |
+-----+
| 3 |
+-----+
```


传入 `JSON_LENGTH` 函数的参数有两个, 其 identifier 分别为 a 和 b

```
mysql> SELECT JSON_LENGTH('{ "a": 1, "b": { "c": 30 } }');
+-----+
| JSON_LENGTH('{ "a": 1, "b": { "c": 30 } }') |
+-----+
| 2 |
+-----+
```

指定路径根的 b 对象下有一个子对象 c

```
mysql> SELECT JSON_LENGTH('{ "a": 1, "b": { "c": 30 } }', '$.b');
+-----+
| JSON_LENGTH('{ "a": 1, "b": { "c": 30 } }', '$.b') |
+-----+
| 1 |
+-----+
```

- `JSON_DEPTH(json_doc)`, 传回指定 JSON 文档的最大（深）阶层，如下。

 mysql> SELECT JSON_DEPTH('{}'), JSON_DEPTH('[]'), JSON_DEPTH('true');

```
+-----+-----+-----+
| JSON_DEPTH('{}') | JSON_DEPTH('[]') | JSON_DEPTH('true') |
+-----+-----+-----+
| 1 | 1 | 1 |
+-----+-----+-----+
```

mysql> SELECT JSON_DEPTH('[10, 20]'), JSON_DEPTH('[[[], {}]]');

```
+-----+-----+
| JSON_DEPTH('[10, 20]') | JSON_DEPTH('[[[], {}]]') |
+-----+-----+
```

```
| JSON_DEPTH('[10, 20]') | JSON_DEPTH('[[[]], {}]') |
```

```
+-----+-----+
```

```
|                2 |                2 |
```

```
+-----+-----+
```

```
mysql> SELECT JSON_DEPTH('[10, {"a": 20}]');
```

```
+-----+
```

```
| JSON_DEPTH('[10, {"a": 20}]') |
```

```
+-----+
```

```
|                3 |
```

```
+-----+
```

- `JSON_CONTAINS_PATH(json_doc, one | all, path[, path...])`, 检索指定的 JSON 文档中是否有一个或全部指定的路径, 如下。

```
mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';
```

第二个参数 'one' 代表 JSON 文档只要能找到任意一个在这个参数之后的路径, 就传回 true

或 1, 变量 j 中的 JSON 文档的“根”下有 a 对象

```
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e');
```

```
+-----+
```

```
| JSON_CONTAINS_PATH(@j, 'one', '$.a', '$.e') |
```

```
+-----+
```

```
|                1 |
```

```
+-----+
```

第二个参数 “all” 代表 JSON 文档要找到所有在这个参数之后的路径, 才会传回 true 或 1,

变量 j 中的 JSON 文档在“根”下没有 e 对象

```
mysql> SELECT JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e');
```

```
+-----+
```

```
| JSON_CONTAINS_PATH(@j, 'all', '$.a', '$.e') |
```

```
+-----+
```

```
|                0 |
```

```
+-----+
```

JSON 文档的“根”下有一对象, 其 identifier 为 d

```
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.c.d');
```

```
+-----+
```

```
| JSON_CONTAINS_PATH(@j, 'one', '$.c.d') |
```

```
+-----+
```

```
|                1 |
```

```
+-----+
```

```
mysql> SELECT JSON_CONTAINS_PATH(@j, 'one', '$.a.d');
```

```
+-----+
```




```
| JSON_CONTAINS_PATH(@j, 'one', '$.a.d') |
+-----+
|                                     0 |
+-----+
```

取出 JSON 文档的内容


取出 JSON 文档内容的功能比较常用, 详细举例如下。

- `JSON_EXTRACT(jond_doc,path)`

```
 # 定义一个名为j的JSON文档的mysql区域变量, 其内容如下
set @j='{ "user":{ "name":{ "first": "abc", "last": "ssss" },
"salary": 12234, "address": [ { "street": "def", "number": "ghi", "phone": "5678890" },
{ "street": "jkl", "number": "mno", "phone": "565656" } ] } }';
#整理JSON文档的内容以阶层方式排列, 比较容易了解JSON文档的结构
#{ "user":
{
  "name": { "first": "abc", "last": "ssss" },
  "salary": 12234,
  "address":
    [ { "street": "def", "number": "ghi", "phone": "5678890" },
      { "street": "jkl", "number": "mno", "phone": "565656" }
    ]
}
}
```

```
 # 验证变量j的内容是否是一个规范的JSON文档
mysql> select json_valid(@j);
```

```
+-----+
| json_valid(@j) |
+-----+
|             1 |
+-----+
1 row in set (0.00 sec)
```

```
 # 查出JSON文档“根”以下的内容
mysql> select json_extract(@j, '$');
```

```
+-----+
+-----+
+-----+
| json_extract(@j, '$') |
```

```
+-----+
+-----+
+-----+
| {"user": {"name": {"last": "ssss", "first": "abc"}, "salary": 12234, "address"
: [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656",
"number": "mno", "street": "jkl"}]}} |
+-----+
+-----+
+-----+
1 row in set (0.00 sec)
```

找出位于JSON文档“根”——“\$”以下成员“user”对象“name”的内容

```
mysql> select json_extract(@j, '$.user.name');
```

```
+-----+
| json_extract(@j, '$.user.name') |
+-----+
| {"last": "ssss", "first": "abc"} |
+-----+
1 row in set (0.00 sec)
```

找出位于JSON文档“根”下成员“user”对象“address”阵列所有的cell


```
mysql> select json_extract(@j, '$.address[*]');
```

```
+-----+
+-----+
| json_extract(@j, '$.address[*]') |
|                                     |
+-----+
+-----+
| [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656",
"number": "mno", "street": "jkl"}] |
+-----+
+-----+
1 row in set (0.00 sec)
```

找出位于JSON文档“根”下成员“address”阵列中位于第二位的cell

```
mysql> select json_extract(@j, '$.user.address[1]');
```

```
+-----+
| json_extract(@j, '$.user.address[1]') |
+-----+
| {"phone": "565656", "number": "mno", "street": "jkl"} |
+-----+
1 row in set (0.00 sec)
```

 # 找出JSON文档位于“根”下的所有成员，和路径“\$”不同的是，它不会列出文件的根——# “user”，而是会列出user之下的所有成员。


```
mysql> select json_extract(@j,'$.*');
```

```
+-----+
+-----+
+-----+
| json_extract(@j,'$.*') |
+-----+
+-----+
| [{"name": {"last": "ssss", "first": "abc"}, "salary": 12234, "address": [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656", "number": "mno", "street": "jkl"}]}] |
+-----+
+-----+
1 row in set (0.00 sec)
```

 # 找出“根”下“user”成员中任何一个地方的子成员“last”的内容


```
mysql> select json_extract(@j,'$.user**.last');
```

```
+-----+
| json_extract(@j,'$.user**.last') |
+-----+
| ["ssss"] |
+-----+
1 row in set (0.00 sec)
```

 # 找出“根”下“user”成员“name”子成员的所有内容

```
mysql> select json_extract(@j,'$.user.name.*');
```

```
+-----+
| json_extract(@j,'$.user.name.*') |
+-----+
| ["ssss", "abc"] |
+-----+
1 row in set (0.00 sec)
```

 # 查出JSON文档“根”成员“user”的“salary”数据类型。请注意函数是可以嵌套的，# 此处先以JSON_EXTRACT找到“salary”对象，再传给JSON_TYPE判断其数据类型

```
mysql> select json_type(json_extract(@j,'$.user.salary'));
```

```
+-----+
| json_type(json_extract(@j,'$.user.salary')) |
+-----+
```

```
+-----+
| INTEGER |
+-----+
1 row in set (0.00 sec)
```

- `JSON_SEARCH(json_doc, one or all, search_str [,escape_char[,path]...])`, 传回指定的 JSON 文档, 或由传入的路径指定部分 JSON 文档下和搜索字符串相符的对象所在的路径, 如果找不到和搜索字符串相符的对象, 则传回 NULL 值。举例如下。

```
# JSON文档从“根”往下找, 第一次碰到值为“abc”的位置为根下阵列中的第一个
# 位置(cell) - ${0}
mysql> SET @j = '["abc", [{"k": "10"}, "def"], {"x": "abc"}, {"y": "bcd"}]';
```

```
mysql> SELECT JSON_SEARCH(@j, 'one', 'abc');
```

```
+-----+
| JSON_SEARCH(@j, 'one', 'abc') |
+-----+
| "${0}" |
+-----+
```

```
# 找出所有值为“abc”的位置, 结果找到两个, 除了根下阵列的第一个位置 (cell) 之外,
# 在第二个位置 (cell) 的成员“x”的内容也是“abc”, 传回的是一个有两个元素的数组
mysql> SELECT JSON_SEARCH(@j, 'all', 'abc');
```

```
+-----+
| JSON_SEARCH(@j, 'all', 'abc') |
+-----+
| ["${0}", "${2}.x"] |
+-----+
```

```
mysql> SELECT JSON_SEARCH(@j, 'all', 'ghi');
```

```
+-----+
| JSON_SEARCH(@j, 'all', 'ghi') |
+-----+
| NULL |
+-----+
```

- `->`, 相当于 `JSON_EXTRACT()`, 会传出整个对象, 例如有一个表 `mycollection` 的 `doc` 字段内含 JSON 文档, 如下。

```
mysql> select doc from mycollection limit 4\G
***** 1. row *****
doc: {"_id": "13a90b0a4af7451c9affb89dae36d5a8", "age": 15, "name": "Sakila"}
***** 2. row *****
```

```

doc: {"_id": "9d0eec5aa65411e6904b0a0027000003", "policy": {"mainInsured":
{"coverageList": [{"item": "ULOB"}]}}}
***** 3. row *****
doc: {"_id": "a34efb1c3344487f90c2d0802aa92098", "age": 39, "name": "Mike"}
***** 4. row *****
doc: {"_id": "c8a5d26905e843dfa2fc15083f6fbd0d", "age": 24, "name": "Susanne"}

# mysql> select json_extract(doc,'$.name') from mycollection limit 4\G
***** 1. row *****
json_extract(doc,'$.name'): "Sakila"
***** 2. row *****
json_extract(doc,'$.name'): NULL
***** 3. row *****
json_extract(doc,'$.name'): "Mike"
***** 4. row *****
json_extract(doc,'$.name'): "Susanne"
4 rows in set (0.00 sec)

```



以 “->” 也能得到同样的结果

```

mysql> select doc->'$.name' from mycollection limit 4\G
***** 1. row *****
doc->'$.name': "Sakila"
***** 2. row *****
doc->'$.name': NULL
***** 3. row *****
doc->'$.name': "Mike"
***** 4. row *****
doc->'$.name': "Susanne"
4 rows in set (0.00 sec)

```

- -> 等同于 JSON_UNQUOTE(JSON_EXTRACT()), 去掉引号再传回, 方便在 WHERE 条件中找出某些信息做比较, 如下。



```

mysql> select doc->'$.name' from mycollection where doc->'$.age' > 20 limit 2;
+-----+
| doc->'$.name' |
+-----+
| "Sunny"      |
| "John"       |
+-----+
2 rows in set (0.00 sec)

```

```
mysql> select doc->>'$.name' from mycollection where doc->'$.age' > 20 limit 2;
+-----+
| doc->>'$.name' |
+-----+
| Sunny        |
| John         |
+-----+
2 rows in set (0.00 sec)
```



注意：由于 JSON_EXTRACT 会传回对象，如果是字符串，则会带双引号，所以必需以 JSON_UNQUOTE() 将双引号移除才能用于比较运算。

建立 JSON 对象

关于建立 JSON 对象，介绍三种类型并举例如下。

- JSON_MERGE(json_doc,json_doc[,json_doc]...), 合并多个 JSON 文档成为一个 JSON 文档。

```
mysql> SELECT JSON_MERGE('[1, 2]', '[true, false]');
+-----+
| JSON_MERGE('[1, 2]', '[true, false]') |
+-----+
| [1, 2, true, false]                  |
+-----+
mysql> SELECT JSON_MERGE('{"name": "x"}', '{"id": 47}');
+-----+
| JSON_MERGE('{"name": "x"}', '{"id": 47}') |
+-----+
| {"id": 47, "name": "x"}                  |
+-----+
```

- JSON_ARRAY(val[,val]...), 将传入的值做成一个 JSON 阵列。

```
mysql> SELECT JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME());
+-----+
| JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME()) |
+-----+
| [1, "abc", null, true, "11:30:24.000000"] |
+-----+
```

- JSON_OBJECT(key,val[,key,val]...), 将传入的一个以上的键（或 identifier）值对做成一个 JSON 对象。

```
mysql> SELECT JSON_OBJECT('id', 87, 'name', 'carrot');
+-----+
| JSON_OBJECT('id', 87, 'name', 'carrot') |
+-----+
| {"id": 87, "name": "carrot"}           |
+-----+
```

更改 JSON 文档

更改 JSON 文档有以下五种类型。


- `JSON_REMOVE(json_doc, path[, path]...)`, 文档中移除由路径所指定的部分文档, 可移除多个路径的对象或以 * 及 ** 指定多个对象, 举例如下。

```
# 从文件中移除“根”下阵列中第二位置的cell
mysql> SET @j = '["a", ["b", "c"], "d"]';
mysql> SELECT JSON_REMOVE(@j, '$[1]');
+-----+
| JSON_REMOVE(@j, '$[1]') |
+-----+
| ["a", "d"]              |
+-----+
```

- `JSON_APPEND(json_doc, path, val[, path, val]...)`, 在指定的阵列后再加上所输入的值, 传回加上该值后的 JSON 文档。
- `JSON_SET(json_doc, path, val[, path, val]...)`, 如果指定的路径没有值, 则会加上输入的值; 如果指定的路径有值, 则用输入的值将其更新, 并传回加上或更新后的 JSON 文档; 如果在文件中没法找到该路径, 则传回 NULL 值。

```
# 文档中位于“$.a”路径的值原来为1, 没有“$.c”路径; JSON_SET(@j, '$.a', 10, '$.c',
# '[true, false]')指定将路径“$.a”的值改为10, 在文件“根”下加一个路径为“$.c”,
# 值为[true, false]的阵列
mysql> SET @j = '{ "a": 1, "b": [2, 3] }';
mysql> SELECT JSON_SET(@j, '$.a', 10, '$.c', '[true, false]');
+-----+
| JSON_SET(@j, '$.a', 10, '$.c', '[true, false]') |
+-----+
| {"a": 10, "b": [2, 3], "c": "[true, false]"}    |
+-----+
```


- `JSON_INSERT()`, 在指定的路径加上输入的值, 并传回加上或更新值后的 JSON 文档, 如果指定的路径已经有值, 则不做任何更改。

 # 文档中“\$.a”路径下已有值，文档“根”下没有路径“\$.c”，JSON_INSERT(@j, '\$.a', # 10, '\$.c', '[true, false]')不会改变文档路径“\$.a”的值，而会在文档“根”下插入 # 一个阵列identifier 'c'，其值为输入的阵列

```
mysql> SELECT JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]');
```

```
+-----+
| JSON_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |
+-----+
| {"a": 1, "b": [2, 3], "c": "[true, false]"}      |
+-----+
```

- JSON_REPLACE(), 更新指定的路径值，并传回加上或更新值后的 JSON 文档，如果找不到指定路径，则不做任何更改。

 # 文件中“\$.a”路径下原来的值为1，文件“根”下没有路径“\$.c”，JSON_REPLACE(@j, '\$.a', 10, '\$.c', '[true, false]')会以10替换在文件根下identifier # 'a'的值，不会在文件“根”下插入identifier 'c'以及该函数所指定的阵列

```
mysql> SELECT JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');
```

```
+-----+
| JSON_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |
+-----+
| {"a": 10, "b": [2, 3]}                               |
+-----+
```

辅助性的函数

还有两个重要的辅助函数，那就是转义符与去转义符。

- JSON_QUOTE(json_val)，如果参数是一个字符串或数值，则在其前后加上双引号，使它成为 JSON 对象，如果字符串中有引号，则传回的结果中会在前后加上一转义符“\”，再加上双引号，如下。

 mysql> SELECT JSON_QUOTE('null'), JSON_QUOTE('"null"');

```
+-----+-----+
| JSON_QUOTE('null') | JSON_QUOTE('"null"') |
+-----+-----+
| "null"             | "\"null\""          |
+-----+-----+
```

```
mysql> SELECT JSON_QUOTE('[1, 2, 3]');
```

```
+-----+
| JSON_QUOTE('[1, 2, 3]') |
+-----+
```



```
| "[1, 2, 3]" |
+-----+
```

- JSON_UNQUOTE(val), 去掉引号和转义符, 只取对象的内容值, 如下。

```
mysql> SET @j = '"abc"';
mysql> SELECT @j, JSON_UNQUOTE(@j);
+-----+-----+
| @j    | JSON_UNQUOTE(@j) |
+-----+-----+
| "abc" | abc               |
+-----+-----+
```

JSON 函数的运用

JSON 函数可以灵活地应用在处理表中存储 JSON 文档的各种场景, 本节用建表时从 JSON 文档中找出指定的数据, 再指定两个表的 JSON 文档内的值做 JOIN, 并更新表中的 JSON 文档为范例说明 JSON 函数的运用。

在 JSON 文档中建虚拟字段和索引以加强性能

在文件型数据库中, 一堆 JSON 文档的集合称为 Collection (本书暂译为“集”)。在 MySQL 中则以表对应 Collection, 存放一堆文件。然而, 要在 JSON 文档集中快速找到符合查询条件的那几个文档 (条件还要用 JSON 函数从文件中找出的值来比对), 须凭借 MySQL 5.7 版以后提供的虚拟字段, 甚至可以用虚拟字段所建的索引, 使查询能够快速找到符合条件的 JSON 文档, 再以 JSON 函数从文件中取出查询所指定的数据。X Protocol 对 Collection 也是以同样的方式查询。这是数据库在底层 InnoDB 引擎中实现的方式。下面举例来说明这个做法。

建一个表 t1, 内含一个 JSON 字段和一个虚拟字段 id。而 id 的值是经由 JSON_EXTRACT 函数算出来的, “STORED” 子句指定虚拟字段的值实际存放在表中是占用空间的字段, 由于把主键加在了这个虚拟字段上, 所以它必须为 STORED。

```
CREATE TABLE t1
(data JSON, id INT AS (JSON_EXTRACT(data,"$.id")) STORED,
PRIMARY KEY(id));
```

如果加上的是非主键, 则可定为不占空间的“VIRTUAL”。在下例中为该“id”虚拟字段加上索引时, 会从每一行存放“data”字段的 JSON 文档中找出各文件“根”下“series”成员的值, 将这个值加到索引 B+ 树的各节点上, 也就是表不实际存 series 的值, 但是索引“series_idx”的节点会存各文件“series”成员的值。

```
ALTER TABLE t1
ADD COLUMN series INT AS (JSON_EXTRACT(data, "$.series")),
ADD INDEX series_idx (series);
```

查询的 WHERE 条件以函数 “JSON_EXTRACT” 找出各行 “data” 字段 JSON 文档中的 “series” 成员的值来比对是否在 3 和 5 之间。

```
SELECT data, series FROM t1 WHERE JSON_EXTRACT(data, "$.series") BETWEEN 3 AND 5;
等价于:
SELECT data, series FROM t1 WHERE series BETWEEN 3 AND 5;
```

传回的结果如下

data	series
{"series": 3, "inverted": 8}	3
{"series": 4, "inverted": 7}	4
{"series": 5, "inverted": 6}	5

由于 JSON_EXTRACT(data, "\$.series") 的值都存在于索引 “series_idx” 上，所以 MySQL 的优化器能利用该索引快速找到符合条件的 JSON 文档是位于哪些行（这点可由 EXPLAIN 该 SQL 的产出证明）。这种以函数索引支持查询的新功能是 5.7 版才出现的。

```
EXPLAIN SELECT data FROM t1 WHERE JSON_EXTRACT(data, "$.series") BETWEEN 3 AND 5\G
```

传回的结果如下。

```
id: 1
select_type: SIMPLE
table: t1
partitions: NULL
type: range
possible_keys: series_idx
key: series_idx
key_len: 5
ref: NULL
rows: 3
filtered: 100.00
Extra: Using where
1 row in set, 0 warnings (0.00 sec)
```

上面的 SQL 命令相当于直接在查询的 WHERE 子句中将 series 字段当成搜寻条件。

```
select `test`.`t1`.`data` AS `data` from `test`.`t1`
where (`test`.`t1`.`series` between 3 and 5)
```

以 JSON 函数更新文件和 JOIN 的条件

下列 t1 和 t2 表都有一个名为 data 的 JSON 字段。想找出 t1 表中所有 JSON 文档“根”下“series”成员的值,和在 t2 表的 JSON 文档“根”下对象“b_series”阵列中第一个位置 (cell) 的值相等的行,并将该行 JSON 文档“根”下名为“inverted”的对象值改为 11 减去 t2 表的 JSON 文档“根”下“b_series”阵列中第一个位置 (cell) 的值,则更新数据的 SQL 命令如下。

```
UPDATE t1, t2
SET t1.data= JSON_INSERT(t1.data,"$.inverted",
11 - JSON_EXTRACT(t2.data,"$.b_series[0]"))
WHERE
JSON_EXTRACT(t1.data, "$.series")
= JSON_EXTRACT(t2.data,"$.b_series[0]");
```

以上说明展示了 MySQL 添加了 JSON 数据类型和 23 个 JSON 函数,使我们能通过传统的 SQL 接口完整地操作 JSON 文档,并能通过索引加快查询性能。同时,由于能将 JSON 文档存于 InnoDB 表中,原来关系型数据库所有的支持事务、多版本并行控制 (MVCC) 及备份监控等功能均能完整保留。

另一方面,数据库管理员也不需要另外适应一套新的数据库技术,让 MySQL 的用户可以同时保有关系型数据库的严谨性,又能拥抱 NoSQL 技术的弹性,以及 JSON 文档不需要做 JOIN 就能查出满足应用需求的数据所带来的性能提升。

然而,MySQL 还不仅止于此。它还更进一步为我们提供了一套全新的 NoSQL 接口——X Protocol,使我们能以纯 NoSQL 的技术开发新的应用。

MySQL X Plugin 和 X Protocol

支持 NoSQL 所做的努力

MySQL 5.7 自第一个正式发行版 (5.7.9) 就加上了新的 JSON 数据类型和 JSON 函数,在朝着支持 NoSQL 的路上迈进了一大步。但是,如果只能通过 SQL 界面开发 NoSQL 应用,还是显得有些削足适履,我们要的是能完全脱离 SQL 的影子,直接以 NoSQL 的方式和数据库互动。所以,MySQLng 项目在原有的 (默认用 3306 端口) SQL 界面之外 (如图 28.1 中的 X Protocol Plugin 部分) 又开发了一个 X Plugin (X 插件),使应用端能通过 X Plugin 的 33060 端口调用 X DevAPI,直接对 MySQL 数据库的 JSON 文档做增删改查等操作。所以,现在的 MySQL 不仅是会说 SQL 语言的关系型数据库,也是一个完整支持 NoSQL 的数据库。

事实上,在 5.6 版本以后 MySQL 就能通过 Memcached 的接口以 key-value 查询的方式和数据库互动 (MySQL memcache 的默认端口为 11211),当查询方式不涉及复杂的 JOIN 和子查询时,通过这个接口绕过 SQL 层,少了处理 SQL 解析和优化的工作,能使性能大幅提升,这也是以 NoSQL 的方式和数据库互动的表现。现在,支持了 JSON 文档界面之后,NoSQL 应用有了更高的可移植性。

安装 MySQL X Plugin

MySQL 默认并不具备处理 X Protocol 的能力,要让 MySQL 能支持 X Protocol 必须先安装 MySQL X Plugin。在安装 X Plugin 之前,MySQL 就只能使用 (默认听 3306) SQL 接口,和熟悉的传统 MySQL 没什么差异 (必需配置插件才能带出 X Protocol,应该是基于降低复杂度和 bug 出现机会的考虑)。

MySQL 自 5.7.12 版以后,可以在软件包的 lib/plugin 目录中找到一个名为 mysqlx.so (Windows 版是 mysqlx.dll) 的文件,就是 X 插件的程序链接库。通过 mysql 客户端程序 (通过默认 3306 端口) 或 MSQL Shell,均可以安装这个插件。

- 用 MySQL 客户端程序安装 MySQL X 插件的命令如下 (不一定要用 root 账户,任何对 mysql.plugin 表有插入权的用户都可以)。

```
$ mysql u<user-name> -hlocalhost P3306 p <password>
$ mysql> INSTALL PLUGIN mysqlx SONAME 'mysqlx.so';
```

- 用 MySQL Shell 安装 MySQL X 插件。
- 在安装好 MySQL Shell 的环境中,做如下操作。

```
$ mysqlsh -u <user-name> -h localhost --classic --dba enableXProtocol
```

可以在 mysql 客户端程序下使用以下命令,在返回的清单中找 mysqlx 以确认 MySQL 的 X 插件是否已装好。

```
mysql> show plugins;
```

当安装好 X 插件之后,就可以准备 X Protocol 的客户端,正式将 MySQL 当成一个全新的文件型数据库了。

MySQL Shell

正如同 MySQL 客户端程序为使用 SQL 接口 (默认 3306 端口) 的人提供方便操作的工具一样,MySQL Shell (如图 28.1 中所示的 MySQL Shell 部分) 是操作 MySQL 文件数据的好帮手,也为各类使用者 (DBA、开发者等) 提供了一个好用的工具,用 NoSQL 的方式可以直接操作数据、管理数据库或做临时性的查询。

安装 MySQL Shell

MySQL Shell 不在 MySQL 软件包内, 需要单独从 dev.mysql.com/downloads/ 上下载其软件包。

如图 28.2 所示, 在 MySQL 下载网页左边的导航列中可以选择 MySQL Shell 选项。有一点需要读者们先注意: 截止到 2016 年底, MySQL Shell 还是第三个 beta (DMR) 版, 在它正式发行之前或许会有一些变动, 或者会加上更多功能。

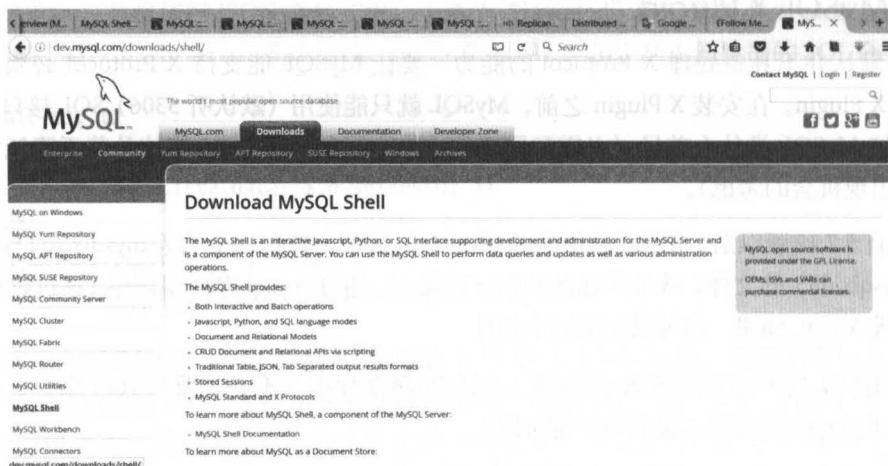


图 28.2

在下载网页 (<https://dev.mysql.com/downloads/shell/>) 上可以根据 MySQL Shell 的运行环境选择合适的版本, 如图 28.3 所示。

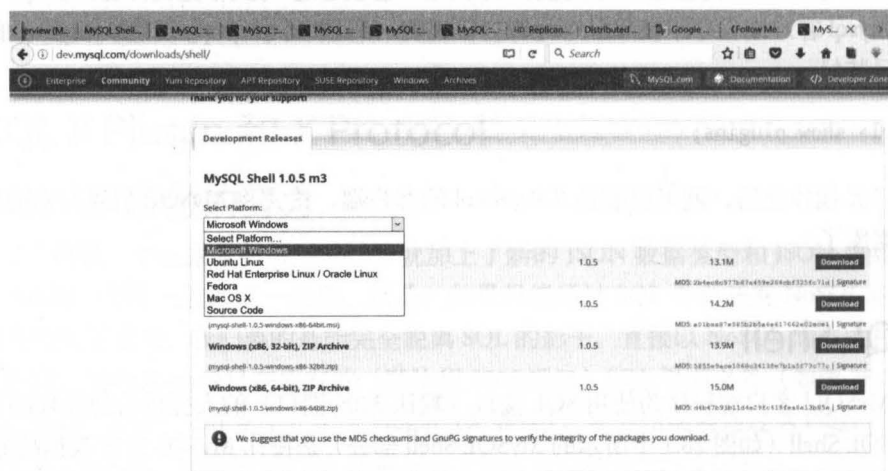


图 28.3

如果选择 ZIP Archive 或 tar 包进行安装，则将其解压，不需安装就可以使用 `mysqlsh` 程序，但需要设定环境的路径变量，将 MySQL Shell 目录下的 `bin` 子目录的路径加到 `PATH` 变量中，让我们在任何目录下都可以激活 `mysqlsh`。根据操作系统的不同，也可以选择 `rpm`、`deb` 或 `msi` 安装包，这些安装包运行时会解开其软件，并加上路径变量或将文件放到适当的位置（例如 `/user/bin`），以方便运行。

运行 MySQL Shell

在操作系统选单中选择 MySQL Shell，或者在终端机命令模式下运行 `mysqlsh` 进入 MySQL Shell，如下。



```
$ mysqlsh
```

```
Welcome to MySQL Shell 1.0.4 Development Preview
```

```
Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.
```

```
Type '\help', '\h' or '\?' for help.
```

```
Currently in Javascript mode. Use \sql to switch to SQL mode and execute queries.  
mysql-js>
```

和 MySQL 客户端程序一样，`mysqlsh` 命令也可以加上参数。例如指定以什么账户连入那个数据库、对话的模式（`X`、`node` 或 `classic`）、语法（`JavaScript`、`Python` 或 `SQL`）或内含批量作业命令的文件。如果要了解有哪些选项，可以用 `--help` 或 `-h` 选项列出 `mysqlsh` 所有的选项及其说明。



```
$ mysqlsh u<user-name> -h<host> -p
```

上列登录数据库的命令看起来是不是很熟悉？使用下列命令指定的 URI 同样能登录数据库。



```
$ mysqlsh uri <user-name>:<password>@<host>:<port>/<db-name>
```

上列命令同样也是通过指定的用户，登录到指定服务器的数据库。进入 `mysqlsh` 后，可输入 `\q` 或 `\quit` 退回到操作系统。

MySQL Shell 有三种对话（Session）模式，如下。

- X Session: 这是默认的互动方式, 这个模式无法执行 SQL 命令, 只能运行 Javascript 或 Python 的命令, 在 MySQL 的 X Session 以后可以发展成同时支持连接多个数据库或数据库分片。
- Node Session: 能执行 SQL、Javascript 或 Python 的命令, 但是 Node Session 只能连接一个数据库实例, 以后也不会扩充成同时连接多个数据库。
- Classic Session: 不用 X Protocol。在这个模式下, 能够使用 SQL 命令, 但开发接口有限, 不支持 NoSQL 的 CRUD 操作, 也不能对 collection 操作。

MySQL shell 有三种命令模供选择, 如下。

- Javascript: 默认的命令模式, 也可以在 mysqlsh 命令中加上 --js 选项进入 Javascript 模式, 提示符为 mysql-js>, 表示能接受 Javascript 的命令。
- Python: 命令加上 --py 选项进入 Python 模式, 提示符为 mysql-py>, 能使 mysqlsh 接受 Python 命令。

```
$ mysqlsh u<user> -h<host> -p --py
```

- SQL: 在 MySQL Shell 下进行 SQL 命令互动, 命令互动方式必须在 MySQL Shell 上指定 classic 或 node session, 再于命令行加上 --sql 选项, 这个模式的接口和传统的 mysql 客户端的操作方式几乎完全相同, 一样是 SQL 命令, 进入此模式的命令如下。

```
$ mysqlsh u<user-name> p h<host> --classic --sql;
```

如同在 mysql 客户端程序下键入 help, 查找 MySQL 客户端程序的命令, 进入 MySQL Shell 后可以用 \? 或 \help 查询 mysqlsh 的操作命令, 所有的 mysqlsh 命令都以 \ 开头, 例如 \c 用来建立数据库联机。

在 MySQL Shell 中操作 JSON 文档

MySQL Shell 为 JSON 文档的操作提供了所需要的命令集, 这些命令都是基于 X DevAPI 的接口而开发出来的, MySQL Shell 的 X DevAPI 的详情请参考 <http://dev.mysql.com/doc/dev/mysqlsh-api-javascript/>。

以 Javascript 命令为例, 以下说明 mysqlsh 在 JSON 文档生命周期中的各种操作命令。

建立 Schema 和更换 Schema

MySQL Shell 有几个默认的对象, 例如 session、db 等, 可以在进入 mysqlsh 以后键入 session 查看目前联机对话的内容, 下例显示目前的互动方式为 X Session, 用 root 账户登录 127.0.0.1 站点, 使用 33060 端口, 如下。

```
mysql-js> session
<XSession:root@127.0.0.1:33060>
```

在 session 对象上建立 Schema 的函数，如下举例建一个名为 mydemo 的 Schema。

```
mysql-js> session.createSchema('mydemo');
<Schema:mydemo>
```

上列命令相当于在 mysql 命令程序上下达 CREATE DATABASE mydemo。

下例可查看之前建立的 mydemo schema 是否存在，以及目前数据库有哪些 schema。

```
schema:
mysql-js> session.getSchemas();
[
  <Schema:information_schema>,
  <Schema:acmug>,
  <Schema:mydemo>,
  <Schema:mysql>,
  <Schema:mysqlnews>,
  <Schema:performance_schema>,
  <Schema:sakila>,
  <Schema:sys>,
  <Schema:world>,
  <Schema:world_x>
]
```

更换使用中的 schema，使用 mysqlsh 命令 \use（或者 \u）即可，如下。

```
mysql-js> \u mydemo
Schema `mydemo` accessible through db.
```

当然，进入 mysqlsh 指定参数时也可以指定建立联机后要用哪一个 database(schema)，如下。

```
$ mysqlsh --uri root@127.0.0.1:33060/mydemo
Creating an X Session to root@127.0.0.1:33060/mydemo
Enter password:*****
Default schema `mydemo` accessible through db.
```

```
Welcome to MySQL Shell 1.0.4 Development Preview
```

```
Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
```



```
owners.
```

```
Type '\help', '\h' or '\?' for help.
```

```
Currently in Javascript mode. Use \sql to switch to SQL mode and execute queries.
mysql-js>
```

上例输入的参数中, URI 的最后一段/mydemo 指定用 mydemo 为联机的默认 schema, 进入 mysqlsh 后响应内容中的 “default schema ‘mydemo’ accessible through db” 说明了这一点。进入 mysqlsh 后, 还可以键入 db 查证一下 mysqlsh 的默认对象 db 是否放在 mydemo 上, 如下。

```
mysql-js> db
<Schema:mydemo>
```

在当前 schema 下操作 Collection 和 Document

在说明 mysqlsh 对 Collection 的操作前, 先简单说明 Document 数据库的基本观念。

一个 Collection 是一堆 JSON 文档的集, 相当于关系型数据库的表。JSON Document 代表 (或描述) 一件事情或一条记录, 相当于关系型数据库在表中的一个 row。

- 建立 Collection

- 通过默认 db 对象的 createCollection 函数, 可以在目前的 schema 下建立 Collection, db 对象代表当下的 session 所在的 schema, 下例为建一个名为 collection1 的 Collection。

```
mysql-js> db.createCollection('collection1');
<Collection:collection1>
```

- 也可以在参数中加上 schema 名称, 该功能使用户可以在任何 schema 下为 mydemo 建立 Collection, 如下。

```
mysql-js> db.createCollection('mydemo.mycollection');
<Collection:mydemo.mycollection>
```

只要有权限在指定的 schema 下新建表的账户, 便均可在位于任何数据库时运行 createCollection, 以参数 <schema-nam>.<collection-name> 的方式为其他的 schema 加上 Collection。

- 加上 Document

- 有了 Collection 以后, 可调用该 Collection 对象的 add 函数将 JSON 文档加入 Collection 中, 如下。

```
mysql-js> db.collection1.add({"name": {"lastname": "Tu", "firstname": "Ivan"},
    "height": 180, "weight": 68, "talents": ["swimming", "dancing", "programming"]});
Query OK, 1 item affected (0.02 sec)
```

- 再加一个 JSON 文档, 如下。

```
mysql-js> db.collection1.add({"name": {"lastname": "Wang", "firstname":
"James"}, "height": 170, "weight": 80, "date of birth": "1990-12-30", "talents":
["make money", "dancing"]});
Query OK, 1 item affected (0.00 sec)
```

- 查询现在已经加了多少文件到 collection1, 如下。

```
mysql-js> db.collection1.find();
[
  {
    "_id": "64b87d23a162d211645db86b2339c6c8",
    "date of birth": "1990-12-30",
    "height": 170,
    "name": {
      "firstname": "James",
      "lastname": "Wang"
    },
    "talents": [
      "make money",
      "dancing"
    ],
    "weight": 80
  },
  {
    "_id": "b85d1476a062d211645db86b2339c6c8",
    "height": 180,
    "name": {
      "firstname": "Ivan",
      "lastname": "Tu"
    },
    "talents": [
      "swimming",
      "dancing",
      "programming"
    ],
    "weight": 68
  }
]
1. documents in set (0.00 sec)
```

如果仔细看上列的输出, 则可以发掘出几个有趣的现象, 如下。

- 虽然前面加 Document 时, 并未在文件中加上 “_id” 对象, 所有的文件都自动加上一个名为 “_id” 的对象, 用于存放自动生成的 uuid 值。

- 文档的结构能自由定义, 不像关系数据库需要事先定一个固定的表结构。上例中的第二个文档比第一个文档多了一个“data of birth”对象, “talents”数组的数目也比第一个少了一个, 这代表 JSON 文档不需要事先定义数据结构和对象名称 (identifier) 就能表达 JSON 文档内各对象的含义。

另一边, 由 MySQL 客户端程序查看数据库在引擎层面发生了什么事。

```
mysql> show create table collection1\G
***** 1. row *****
      Table: collection1
Create Table: CREATE TABLE `collection1` (
  `doc` json DEFAULT NULL,
  `_id` varchar(32) GENERATED ALWAYS AS (json_unquote(json_extract(`doc`, '$._id'))
  STORED NOT NULL,
  PRIMARY KEY (`_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

上例显示每一个 collection 都对应一个 InnoDB 表。表有两个字段——doc 字段的数据类型是 json; 另一个 _id 字段是由 json_extract 函数生成的 STORED (自函数算出值实际存放于表中, 以后查询时不需要再算一次) 字段, 它的值是在 doc 字段中找到的名为 _id 的物件之值, 并且将主键加在这个 _id 字段上。

- 添加普通索引。

也能为 Collection 的 weight 加上 (非唯一) 普通索引, 如下。

```
mysql-js> db.collection1.createIndex("weightIdx").field("weight", "INTEGER",
false).execute();
Query OK (0.06 sec)
```

另一方面, 加上索引后的 InnoDB 表结构如下所示。

```
mysql> show create table collection1\G
***** 1. row *****
      Table: collection1
Create Table: CREATE TABLE `collection1` (
  `doc` json DEFAULT NULL,
  `_id` varchar(32) GENERATED ALWAYS AS (json_unquote(json_extract(`doc`, '$._id'))
  STORED NOT NULL,
  `$ix_i_2D8FA3E83FC26B3A572C94D67A6F56CAC0CA1EAB` int(11) GENERATED ALWAYS AS
  (json_extract(`doc`, '$.weight')) VIRTUAL,
  PRIMARY KEY (`_id`),
  KEY `weightIdx` (`$ix_i_2D8FA3E83FC26B3A572C94D67A6F56CAC0CA1EAB`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

由此证明, X DevAPI 会因为这个 collection1.createIndex 函数在数据库的 collection1 表上多加一个名为 \$ix_i_<uuid> 的虚拟生成字段, 其值是由 json_extract 文件中的 weight 对象取出的值所构成的, 并且在其上加了索引-weightIdx。

- 进行条件查询。

也可以对 collection 进行条件查询, 甚至动态 定值于条件中, 以下为从 Collection 中找出 weight > 70 的文件的例子。

```
mysql-js> db.collection1.find("weight > :kg").bind("kg",70);
[
  {
    "_id": "64b87d23a162d211645db86b2339c6c8",
    "date of birth": "1990-12-30",
    "height": 170,
    "name": {
      "firstname": "James",
      "lastname": "Wang"
    },
    "talents": [
      "make money",
      "dancing"
    ],
    "weight": 80
  }
]
1 document in set (0.00 sec)
```

由于查询条件里有 weight 对象, 执行计划可能会用到前一段所建立的 weightIdx 的索引。

- 指定查询结果中的对象。还可以指定要由查询的结果中只传回指定的对象, 下例显示不需要看整份 JSON 文档, 只要传回 name 和 height 对象时可以 fields 函数指定要传回的对象即可。

```
mysql-js> db.collection1.find("weight > 70").fields(["name","height"]);
[
  {
    "height": 170,
    "name": {
      "firstname": "James",
      "lastname": "Wang"
    }
  }
]
```

```

    }
  ]
  1 document in set (0.00 sec)

```

- 删除 collection。

最后不需要这个 collection 时, 可以删除这个 collection, drop collection 需要用 session 对象的 dropCollection 函数, 它有两个必要的参数——schema 名称和 collection 名称。删除 collection 后, 该 schema 就查不到它了。代码如下。

```

mysql-js> session.dropCollection("mydemo","collection1");
Query OK (0.02 sec)

mysql-js> db.getCollections()
[
]

```

用脚本执行 MySQL Shell

就如 SQL 可以通过编写脚本在 mysql 客户端程序运行批量作业一样, 也可以编写一套 Javascript 或 Python 脚本在 MySQL Shell 上运行批量作业, 举例如下。

建立一个名为 Transaction-sample.js 的脚本, 这个脚本会运行 1000 次循环, 每次循环插入五个 JSON 文档, 每个循环都是一个事务, 也就是一个事务提交五个 JSON 文档。

```

/* 引入mysqlx模块, 本地变量-mysqlx指向mysqlx对象, 由mysqlx的getSession函数建立
   数据库联机 */
var mysqlx = require('mysqlx').mysqlx;
/* 建立数据库联机 */
var session = mysqlx.getSession( {
  host: 'localhost', port: 33060,
  dbUser: 'root', dbPassword: '<password>' } );
/* 将schema移到my_test */
var db = session.getSchema('my_test');
/* 在my_test schema下建立一个新的 collection——mycollection, 如果该collection在此
   之前已经存在, 则先drop它 */
session.dropCollection('my_test','mycollection');
db.createCollection('mycollection');
/* 取一个collection对象myColl, 各JSON文档均将其add函数加入collection中 */
var myColl = db.getCollection('mycollection');
var iter;
/* 循环运行1000次, 每次先以session.startTransaction()定事务的起始点。循环插入5个
   文件, 如果运行中间有错误或强制中断, 则之前提交的事务会存在数据库中, 做到中间

```

的事务会回滚。所以collection中存在的文件数目会是5的倍数（实现事务的数据完整性）。由此证明，X Dav API是支持事务和数据完整性的。最后以session.commit()结束并提交事务 */

```
try {
  for (iter = 0; iter < 1000; iter++) {
    session.startTransaction();
    myColl.add({name: 'Sunny', age: 23, height: 1.3, weight: 73.3}).execute();
    myColl.add({name: 'Albert', age: 24, height: 1.4, weight: 74.3}).execute();
    myColl.add({name: 'Bred', age: 25, height: 1.5, weight: 75.3}).execute();
    myColl.add({name: 'Connie', age: 26, height: 1.6, weight: 76.3}).execute();
    myColl.add({name: 'David', age: 27, height: 1.7, weight: 77.3}).execute();
    /* Commit the transaction if everything went well */
    session.commit();
    print('Data inserted successfully.\n');
  }
}
catch (err) {
  /* 如果有错误，则回滚未完成的事务 */
  session.rollback();
  /* Printing the error message */
  print('Data could not be inserted: ' + err.message);
}
```

将上列脚本送到 MySQL Shell 运行，结果如下。

```
$ mysqlsh -f transaction-sample.js
Data inserted successfully.
Data inserted successfully.
...
Data inserted successfully.
```

最后会出现 1000 次 Data inserted successfully。

登录查看运行的结果，如下。

```
mysql-js> db.mycollection.find()
[
  {
    "_id": "ffe5ac80c362d2114072b86b2339c6c8",
    "age": 24,
    "height": 1.4,
    "name": "Albert",
    "weight": 74.3
```

```

    },
    {
        "_id": "fffaab7ec362d2114072b86b2339c6c8",
        "age": 27,
        "height": 1.7,
        "name": "David",
        "weight": 77.3
    }
    ...
]
5000 documents in set (0.00 sec)

```

上面的例子中，建数据库连接不是用 Connection 对象，而是 Session 对象。在 Connection 上引进 Session 的概念，也透露着 X DevAPI 将朝向支持一个 Session 可能包含多个 Connection，每个 Connection 分别连到不同的数据库，而能支持数据库分片（data shard）的方向发展。

X DevAPI

Connector 实现了 X DevAPI，来支持多种应用程序便于 JSON 和 MySQL 互动。至此，整个 *MySQL as a Document Store* 的技术拼图只剩下了最后一片——使应用层能通过 X Protocol 运行 CRUD 的函数，（如图 28.1 所示的对应部分）直接操作位于数据库内的 JSON 文档。

MySQL 在各种新版本的应用程序链接器（包括 Connector/J、Connector/NET、Connector/C++ 等）上增加了新的支持 X Protocol 的模块，让应用程序能直接调用这些模块中实现 X DevAPI 的接口函数，通过 X Protocol 直接使用和更新数据库的 JSON 文档。这些链接器还在持续完善中，目前支持 X Protocol 的链接器有 Connector/J、Connector/Python、Connector/NET、Connector/C++ 和 Connector/Node.js。

本章以一个 Java 应用程序——MySQLStore.java（这个 Java 程序会插入三个 JSON 文档到 my-collection 中，并找出 name 为 Sakila、age 小于 20 的文件，并列出来）为例，来演示如何调用 Connector/J 的 X DevAPI 接口。

有兴趣测试这个功能的读者可以按以下步骤配置 Java 运行环境，并编写 Java 程序来体验一下 Document based 应用。

1. 准备 JSON 文档数据库的 Java 开发环境。

开发及运行 MySQLStore Java 程序除了要有 JDK，还需要合适的 Java 驱动器。驱动器 Connector/J 6.0 以后的版本才能支持 JSON 文档数据库。在 MySQL 的软件下载网页（网址为 dev.mysql.com/downloads/Connector/J/），点选“Development Releases”页签，才能找到 Connector/J 6.0.x 版的.jar 包。下载完这里的压缩文件后，做以下三个操作，基本上就完成了开发环境的准备了。

- 解压缩该文件。
- 找到 mysql-connector-java-6.0.<x>-bin.jar 文件。
- 并将 java.jar 文件的位置添加到操作系统环境变量——CLASSPATH 中，或者在 Java IDE 中添加一个链接库给这个.jar 文件。

2. MySQL 文档数据应用的 Java 程序。

```

package mysqldocstore;
import com.mysql.cj.api.x.*;
import com.mysql.cj.x.MysqlxSessionFactory;
import com.mysql.cj.x.json.*;
/* 以上import的三个package/class都是MySQL Connector/J 6.0.x 新增的 */
public class MySQLdocStore {
    public static void main(String[] args) {
        String url = "mysqlx://localhost:33060/my_test?user=itu&password=<password>";
        /* 通过mysqlx建立数据库对话 (Session) 对象，数据库主机位于localhost */
        /* 数据库监听33060端口，用户名为itu */
        /* schema为my_test (可不在URI指定Schema) */
        /* <password>要替换成数据库的root密码 */
        XSession mySession = new MysqlxSessionFactory().getSession(url);
        /* 开启一个事务 */
        mySession.startTransaction();
        /* 将位置移到your_test schema */
        Schema myDb = mySession.getSchema("your_test");
        /* 找一个名为mycollection的Collection，并以myColl变量指向该物件 */
        Collection myColl = myDb.getCollection("mycollection");
        /* 插入三个JSON文档 */
        myColl.add("{\"name\":\"Sakila\", \"age\":15}").execute();
        myColl.add("{\"name\":\"Susanne\", \"age\":24}").execute();
        myColl.add("{\"name\":\"Mike\", \"age\":39}").execute();
        /* 提交事务 */
        mySession.commit();
        /* 在mycolleciton中找文件，条件是name是“Sakila”字符串，且age小于20 */
        DocResult docs = myColl.find(name like :name AND age < :age)
            .bind("name", "Sakila").bind("age", 20).execute();
        /* 取出查找到的文件，以doc变量指向该对象，如果找不到符合条件的文件，
           则返回NULL */
        DbDoc doc = docs.fetchOne();
        /* 列出查询到的JSON文档 */
        System.out.println("Age below 20 is "+doc);
    }
}

```


3. 运行结果。

由于已经将 MySQLdocStore.class 和 mysql-connector-java-6.0.<x>-bin.jar 一同压缩在.jar 包中以方便布署, 所以运行时要指定 MySQLdocStore.jar 文件。运行后, 在数据库中会插入三个 JSON 文档, 最后列出其中一个 JSON 文档, 其 age 对象的值小于 20。

```
$ java -jar MySQLdocStore.jar
Age below 20 is {
  "_id" : "bb8c3bf0242a44a0a49767e3dcee7cc9",
  "age" : 15,
  "name" : "Sakila"
}
```

通过 MySQL Shell 查看 mycollection Collection 可以看到, 在 mycollection 中有三个 JSON 文档都是通过 MySQLdocStore.class 程序插入的, 如下。

```
mysql-js> db.mycollection.find();
[
  {
    "_id": "2f5a69049e0744f4b1f3abd9bbbdd2dd",
    "age": 39,
    "name": "Mike"
  },
  {
    "_id": "bb8c3bf0242a44a0a49767e3dcee7cc9",
    "age": 15,
    "name": "Sakila"
  },
  {
    "_id": "d9d3e1c86a3643be93e65ac1981c659f",
    "age": 24,
    "name": "Susanne"
  }
]
3. documents in set (0.00 sec)
```

在 SQL 界面上查询 mycollection 表, 也可以在 mycollection 表中看到这些行含有 JSON 文档, 如下。

```
mysql> select * from mycollection\G
***** 1. row *****
doc: {"_id": "2f5a69049e0744f4b1f3abd9bbbdd2dd", "age": 39, "name": "Mike"}
_id: 2f5a69049e0744f4b1f3abd9bbbdd2dd
***** 2. row *****
```

```

doc: {"_id": "bb8c3bf0242a44a0a49767e3dcee7cc9", "age": 15, "name": "Sakila"}
_id: bb8c3bf0242a44a0a49767e3dcee7cc9
***** 3. row *****
doc: {"_id": "d9d3e1c86a3643be93e65ac1981c659f", "age": 24, "name": "Susanne"}
_id: d9d3e1c86a3643be93e65ac1981c659f
3 rows in set (0.00 sec)

```

总结

笔者认为 MySQL 支持 JSON 文档数据库是 MySQL 近年来最重要的创新之一，这套技术促成了 MySQL 将 NoSQL 和关系数据库的优势熔于一炉，让广大 MySQL 用户能在拥抱新的 NoSQL 技术的同时，不需要去学习另外一套新的数据库，去磨合陌生的新技术。用户可以在他们已经熟悉的 MySQL 上使用 Schema Less 数据结构支持 Dev/Ops 的开发方法，进而加速应用程序的开发，满足应用业务上占得市场先机的同时，又能享有关系数据库已经很成熟的数据完整性、支持事务、MVCC（多版本并行控制）和崩溃回复的优点。

另一方面，DBA 们也能使用和原来 MySQL 相同的工具为数据库进行备份、调优、监控和安全管制。MySQL 这项与时俱进的发展使得 MySQL 所有的从业人员都能轻松应对 IT 大环境的改变，让 MySQL 应用开发者、DBA 和应用业务拥有者三方都能同获其益。

对开发者而言，除了 MySQL 数据库在从核心到 MySQL Shell 工具和各种应用的 Connector 上推出支持 JSON 的功能外，还需要在应用开发环境上提供更丰富的框架，以进一步使 MySQL 的开发者能够快速而方便地开发应用。由于许多 Connector 技术（例如 Java、Node.js 等）已经有很多现成的框架支持 JSON，因此让我们在这方面的发展有了扎实的基础。例如使用 Node.js 的开发者或许知道（或用过）MEAN（MondoDB、Express、Angularjs、Node.js）框架，它能使开发者更高效、更容易地开发互联网应用。经过 Connector/Node.js 开发人员的努力，MySQL 现在也支持 MEAN 框架，只是组合的含义变为 MySQL、Ecpress、Angular.js 和 Node.js（相关信息和使用案例可参考 <http://insidemysql.com/develop-by-example-document-store-working-with-express-js-angularjs-and-node-js/>）。相信随着 MySQL as a document store 功能的持续完善，将有更多的框架和 IDE（整合开发环境）可以支持这个技术堆栈，或许 MySQL 会在 LAMP 之后为 IT 界带来另一波高潮。

参考资料

- [1] MySQL 研发团队博客, Using MySQL As a Document Store, <http://mysqlserverteam.com/mysql-5-7-12-part-6-mysql-document-store-a-new-chapter-in-the-mysql-story/>.
- [2] JavaScript X DevAPI Tutorial, <http://dev.mysql.com/doc/refman/5.7/en/mysql-shell-tutorial-java>

script.html.

[3] X DevAPI 使用者手册,<http://dev.mysql.com/doc/dev/connector-j/>.

[4] MySQL Shell 使用者手册,<https://dev.mysql.com/doc/refman/5.7/en/mysql-shell.html>.



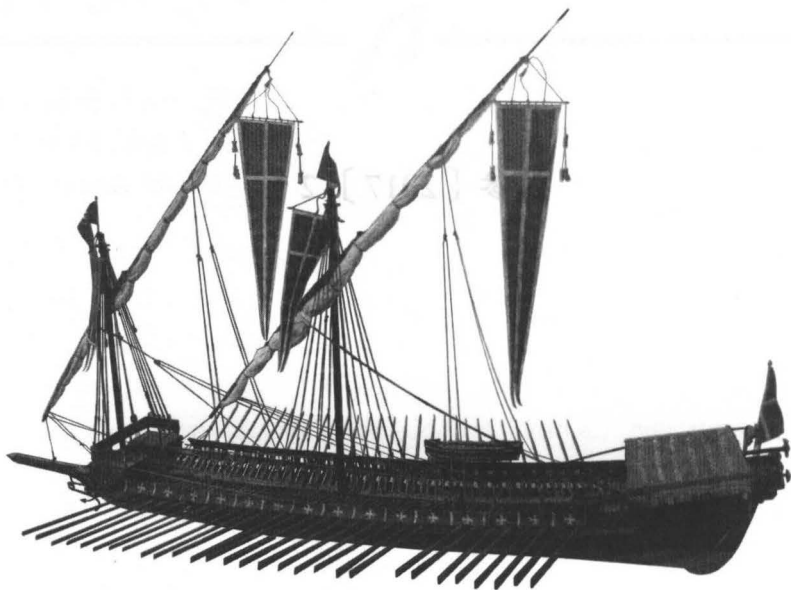
第二部分 Galera 篇



内参〔2017〕2号

Galera 是意大利语, 它的意思是 Galley。那么 Galley 又是什么意思呢? 这是一种桨帆船, 如下图所示, 音译为贾列船, 在公元前 1000 年后出现, 是以人力划船作为主要动力的船种。这种船的主要特点是船体巨大、修长, 人力为主要动力。它有一到三层桨, 每层又有很多很多船桨并排, 船的主要动力就是靠数量巨大的桨手共同滑动船桨获得, 桨手的动作对于船的行进非常重要, 桨手们需要保持高度的一致性和协调性, 一致行动, 并行出力, 才能获得最大动力。在 Galera 的官方文档中, 对它有如下这样的描述。

The word galera is the Italian word for galley. The galley is a class of naval vessel used in the Mediterranean Sea from the 2nd millennium B.C.E. until the Renaissance. Although they used sails when the winds were favorable, their principal method of propulsion came from banks of oars. In order to manage the vessel effectively, rowers had to act synchronously, lest the oars become intertwined and get blocked. Captains could scale the crew up to hundreds of rowers, making the galleys faster and more maneuverable in combat.



Galera 的作者分别是 SEPPO JAAKOLA、ALEXEY YURCHENKO 和 TEEMU OLLAKKA, 他们有大约 20 多年的数据库集群化产品开发和经验, 在开发 Galera 之前, 他们都在 Continuent 公司工作, Continuent 早期为 MySQL 提供了另外一个著名的工具 Tungsten Replicator。在作者的自述中说到, 他们多年来一起在数据库和数据集群领域工作, 彼此都非常熟悉, 经常会聚在一起谈技术, 谈工作, 谈碰到的一些技术方案的缺陷和不足。在这样的交谈中, 意识到相互之间有很多共同点, 觉得应该做点事情, 做点更美好的事。于是, 在 2007 年, 他们发布了 Galera Cluster for MySQL, 一款崭新的、高性能、易扩展的开源数据库集群化解决方案。给自己的产品取名为 Galera, 很显然是取其良好的并发性和一致性的特点, 这也是

Galera 的优势所在。Galera Cluster 的主要用途是为 MySQL 提供一致性的集群化解决方案，以一个 dlopenable 库的形式提供给 MySQL，并通过自身的 Write-Set 提供复制服务，从而实现 MySQL 的多线程并行复制和多源复制。此外，它自带集群节点管理机制，可以主动监测集群节点状态，自动管理有问题的数据节点，同时也可以实现集群的多点写入和平滑扩容。Galera Cluster 最关注的是数据的一致性，对待事务的行为时，要么在所有节点上执行，要么都不执行，它的实现机制决定了它对待一致性的行为非常严格，这也能非常完美地保证 MySQL 集群的数据一致性。

我们从 2012 年就开始关注 Galera，在众多的 MySQL 开源项目中，这是一个非常有特色的产品。在 2013 年之后，我们接手了去哪儿网的 MySQL 数据库管理工作。去哪儿网是一个典型的电子商务网站，提供机票、酒店及其他旅游相关的服务，所以订单和库存数据非常重要，这就对数据库的高可用性和数据一致性提出了更高的要求。经过长期研究，不断地试错，终于在 Galera 的基础上，实现了一套自己的 MySQL 解决方案，截止到现在，已经有非常庞大的线上集群运行着 Galera。在这个过程中，也积累了很多 Galera 的技术经验，希望这些经验也能帮助其他 Galera 使用者解决疑难或规避问题。

目前，对 Galera Cluster 的封装有两个，虽然名称不同，但实质都是一样的，使用的都是 Galera Cluster。一个是 MySQL 的创始人 Monty 在自己全新的 MariaDB 上实现的 MariaDB Cluster，一个是著名的 MySQL 服务和工具提供商 Percona 实现的 Percona Xtradb Cluster，简称为 PXC。

29

Galera Cluster 的设计与实现

前面已经介绍过，Galera 的产品以 Galera Cluster 的方式提供给 MySQL 或类 MySQL 的 MariaDB，为 MySQL 提供集群化解决方案。到目前为止，Percona 所封装的 Percona Xtradb Cluster 方案最为成熟。众所周知，去哪儿网的业务类型属于电子商务，业务中的各种订单交易、库存管理占了很大一部分，而这些需求都是用 MySQL 作为底层数据存储来解决的。面对业务对数据的一致性和可用性的需求，最近几年以来，引入了基于 Percona Xtradb Cluster（以下简称 PXC）的高可用数据库解决方案。在接下来的内容里，将详细介绍 Galera Cluster 的特点。

Galera Cluster 的优点

首先来看一幅图（摘自 Percona XtraDB Cluster 官方文档：<https://www.percona.com/doc/percona-xtradb-cluster/5.7/intro.html>），如图 29.1 所示。

图 29.1 中有三个实例，组成了一个集群，而这三个节点与普通的主从架构不同，它们都可以作为主节点，三个节点是对等的，一般称为 multi-master 架构。当有客户端要写入或读取数据时，随便连接哪个实例都是一样的，读到的数据是相同的，写入某一个节点之后，集群自己会将新数据同步到其他节点上。这种架构不共享任何数据，是一种高冗余架构。

一般的使用方法是，在这个集群上面，再搭建一个中间层，这个中间层的功能包括建立连接、管理连接池，负责使三个实例的负载基本平衡，并能够在客户端与实例的连接断开之后进行重连，还可以进行读写分离（在机器性能不同的情况下，可以做这样的优化）等。使用这个中间层之后，由于这三个实例的架构在客户端方面是透明的，客户端只需要指定这个集

群的数据源地址，连接到中间层即可，中间层会负责客户端与服务器实例连接的传递工作。由于这个架构支持多点写入，所以完全避免了主从复制经常出现的数据不一致的问题，从而可以做到主从读写切换的高度优雅，在不影响用户的情况下，完成离线维护等工作。MySQL 的高可用从此开始，非常完美。

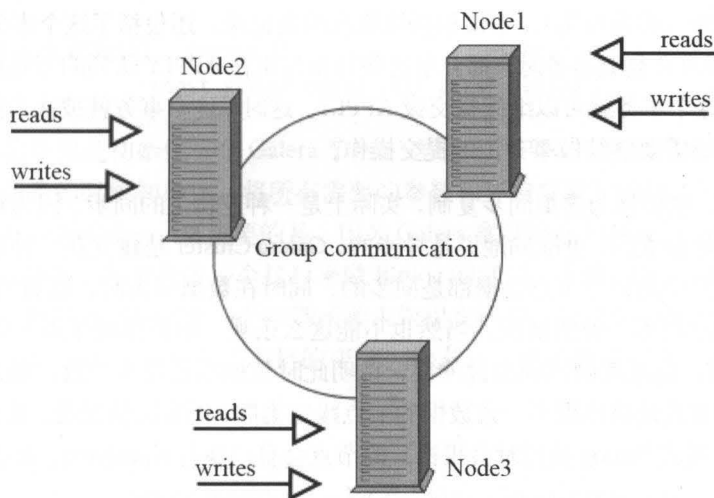


图 29.1

这里最核心的问题其实是，在三个实例之间，因为它们的关系是对等的，是 multi-master 架构的，所以要考虑在同时写入的时候，如何保证整个集群数据的一致性、完整性与正确性的问题。

在 MySQL 通常的使用过程中，也不难实现一种 multi-master 架构，但是一般需要上层应用来配合，比如先要约定每个表必须要有自增列，并且如果是 2 个节点，一个节点只能写偶数值，而另一个节点只能写奇数值，同时 2 个节点之间互相做复制，因为 2 个节点写入的东西不同，所以复制不会冲突。在这种约定之下，可以基本实现多 master 的架构，也可以保证数据的完整性与一致性。但这种方式使用起来还是有限制，同时还会出现复制延迟，并且不具有扩展性，不是真正意义上的集群。

Galera 的引入

PXC 是在 MySQL Server 基础上，封装了 Galera 而实现的具有高一致性、支持多点写入的同步复制集群架构。通过封装 Galera 而实现的 MySQL Cluster 称为 Galera Cluster，目前包括 Percona Xtradb Cluster 和 Mariadb Cluster。Galera 是建立在 MySQL 同步基础之上的，使用 Galera Cluster 的时候，应用程序可以直接读、写某个节点的最新数据，并且可以在不影响应用程序读写的情况下，使某个节点下线，因为支持多点写入，使得 Failover 变得非常简单。

所有的 Galera Cluster，都对 Galera 提供的接口 API 做了封装。这些 API 为上层提供了丰富的状态信息及回调函数，通过这些回调函数，做到了真正的多主集群、多点写入及同步复制，这些 API 被称作 Write-Set Replication API，简称为 wsrep API。

通过这些 API，Galera Cluster 提供了基于验证的复制，是一种乐观的同步复制机制，一个将要被复制的事务（称为写集），不仅包括被修改的表记录，还包括了这个事务产生的所有 Binlog，每一个节点在复制事务时，都会拿这些写集与正在 APPLY 队列的写集做比对，如果没有冲突的话，这个事务就可以继续提交或 APPLY。这时，这个事务就被认为提交了，之后在数据库层面，还需要继续做事务上的提交操作。

这种方式的复制，也被称为虚拟同步复制，实际上是一种逻辑上的同步。因为每个节点的写入和提交操作还是独立的，更准确地说是异步的，Galera Cluster 是建立在一种乐观复制的基础上的，假设集群中的每个节点数据都是同步的，同时在数据写入时，都有写集验证机制，所以理论上不会出现不一致的情况。当然也不能这么乐观，如果出现了不一致，比如主库（相对）插入成功，而从库则出现主键冲突，说明此时数据库已经不一致，那么此时 Galera Cluster 所采取的方式是将出现不一致数据的节点踢出集群，而实际情况是，执行时报错（数据不一致时 Row 模式 Binlog 执行时会报错）的节点会自己执行 shutdown，从而退出集群。

然而，使用 Galera 会在内部通过检查键值冲突的机制实现真正意义上的 Multi-Master。下面就这个问题展开讨论，并深入解析它的实现方式。

现在已经知道，Galera Cluster 对 MySQL 生态中的高可用方面有着非常重要的提升作用。目前，Galera Cluster 具备的功能包括如下七个方面。

- 多主架构：真正的多点读写集群，在任何时候读写的的数据都是最新的。
- 同步复制：集群不同节点之间的数据同步，没有延迟，在数据库挂掉之后，数据不会丢失。
- 并发复制：从节点在 APPLY 数据时，支持并行执行，有更好的性能表现。
- 故障切换：因为支持多点写入，所以在出现数据库故障时可以很容易地进行故障切换。
- 热插拔：在服务期间，如果数据库挂了，只要监控程序发现得够快，不可服务时间就会非常少。在节点故障期间，节点本身对集群的影响非常小。
- 自动节点克隆：在新增节点或停机维护时，增量数据或基础数据不需要人工手动备份提供，Galera Cluster 会自动拉取在线节点数据，集群最终会变为一致。
- 对应用透明：集群的维护，对应用程序是透明的，几乎感觉不到。

以上七点，足以说明 Galera Cluster 是一个既稳健，又在数据一致性、完整性及高性能方面有出色表现的高可用解决方案。

下面以 PXC 使用 Galera 的实现为代表，讲述一下 Galera Cluster 的一般实现方式。首先从接口说起。

Galera 接口

Galera 提供了很多接口，下面是几个最重要的接口。

galera_init

```
wsrep_status_t (*init) (wsrep_t* wsrep,
                        const struct wsrep_init_args* args);
```

上面这个接口的作用是初始化一个 Galera 节点，这是一个 PXC 节点调用的第一个 wsrep 接口，在启动服务器的时候初始化，将所有需要的参数和环境变量初始化，如：将集群名字、实例地址等信息告诉 Galera。更重要的是，因为 Galera 拿到的是 Binlog，它是不知道如何复制数据的，所以这里必须要指定一个接口来做 Binlog 的复制。当然，Galera 也并不知道数据库如何做提交，所以也同样需要一个回调函数来做提交操作。而这个 galera_init 的接口就是负责在启动时告诉 Galera 一些关于上层的变量、状态及回调函数等信息，具体工作如下代码所示。

```
struct wsrep_init_args wsrep_args;
struct wsrep_gtid const state_id = { local_uuid, local_seqno };
wsrep_args.data_dir          = wsrep_data_home_dir;
wsrep_args.node_name         = (wsrep_node_name) ? wsrep_node_name : "";
wsrep_args.node_address      = node_addr;
wsrep_args.node_incoming     = inc_addr;
wsrep_args.options           = (wsrep_provider_options) ? wsrep_provider_options : "";
wsrep_args.proto_ver         = wsrep_max_protocol_version;
wsrep_args.state_id          = &state_id;
wsrep_args.logger_cb         = wsrep_log_cb; //底层通过上层MySQL的日志系统来写日志
wsrep_args.view_handler_cb   = wsrep_view_handler_cb;
                                //初始化状态的，决定要不要做state transfer
wsrep_args.apply_cb          = wsrep_apply_cb; //从节点应用接收到的Binlog的方法
wsrep_args.commit_cb         = wsrep_commit_cb; //事务提交
wsrep_args.unordered_cb      = wsrep_unordered_cb;
wsrep_args.sst_donate_cb     = wsrep_sst_donate_cb; //Donor节点提供数据的方法
wsrep_args.syncd_cb          = wsrep_syncd_cb;
rcode                        = wsrep->init(wsrep, &wsrep_args);
                                //调用接口，告诉Galera这些信息
```


内容解释如下。

- state_id: 状态信息，也叫作 GTID 值，包括一个 UUID 和一个 seqno。这个值是当前节点启动时的一个启动 GTID，表示当前节点最新的 GTID 值，在加入集群时，从这个点

开始向其他节点寻求 Donor (增量数据)。在新版本的 PXC 中, 这个值会自动从 Innodb 文件页面中取出来, 而如果文件 grastate.dat 中的 seqno 为 -1, 也没有关系, 因为内部就可以保证 seqno 为合法值。当然, 如果 UUID 为非法, 就还会选择做 SST。

- `wsrep_data_home_dir`: 默认就是 MySQL 的 `datadir` 值, 可以设置一个其他值。这个目录是用来存储 `grastate.dat`、`gcache` 等文件的。
- `wsrep_node_name`: 对应参数 `wsrep_node_name`。
- `wsrep_provider_options`: 对应参数 `wsrep_provider_options`。这里可以设置 Galera 底层的一些参数值, 一般情况下, 只需要设置下面这些即可: `evs.keepalive_period=PT1S`; `evs.inactive_check_period=PT0.5S`; `evs.suspect_timeout=PT5S`; `evs.inactive_timeout=PT15S`; `gcs.fc_factor=1.0`; `gcs.fc_limit=1024`; `gcs.fc_master_slave=yes`; `gcache.size=10K`; `cert.log_conflicts=yes`; `gmcast.listen_addr=tcp://0.0.0.0:4106`; `ist.recv_addr=localip:port`; 具体这些参数有什么意义, 第 30 章将会专门对一些重要的参数进行讲述。
- `wsrep_log_cb`: 回调函数, 用来做日志的。因为 Galera 产生一些日志后, 需要告诉 MySQL 转换为 MySQL 的日志, 所以就需要通过这个接口来告诉 MySQL, 让 MySQL 记录底层所产生的日志。
- `wsrep_view_handler_cb`: 回调函数, 用来做 SST/IST 的。第 34 章会对其进行详细讲述。
- `wsrep_apply_cb`: 回调函数, 用来告诉 Galera 如何将写集复制到从节点。下面会讲述。
- `wsrep_commit_cb`: 回调函数, 用来告诉 Galera 如何做事务提交。
- `wsrep_sst_donate_cb`: 回调函数, 专门用来给 Joiner 提供 SST 数据。

galera_connect

```
 wsrep_status_t (*connect) (wsrep_t*      wsrep,
                             const char *  cluster_name,
                             const char *  cluster_url,
                             const char *  state_donor,
                             wsrep_bool_t bootstrap);
```

这个接口, 是继上一个接口之后, 第二个调用的接口。这个接口的作用是将当前节点加入集群中。但在这个操作中, 会通过回调函数 `wsrep_view_handler_cb` 来判断新加入节点与集群的数据是否同步。如果不同步, 就再通过不断地消息交互, 做 SST 或 IST 操作 (关于 state transfer, 请参考第 34 章)。这个接口需要指定连接到集群的名字, 同时还需要通过参数 `bootstrap` 指定是不是集群中的第一个节点, 对应 PXC 参数为 `--wsrep-new-cluster`, 接口执行完成之后, 相应的集群基本上就在正常运转了。

galera_recv

```
wsrep_status_t (*recv)(wsrep_t* wsrep, void* recv_ctx);
```

galera_recv 的作用是，在这个函数里阻塞式地接收其他节点及本节点发送的数据，并且调用复制 APPLY 函数执行复制操作。这个函数只要返回相应的值，就说明执行出错或发生了其他问题。而只要执行正常，这个函数就会一直阻塞在这里（具体应该不是阻塞，而是一直在 recv 函数中执行操作），而里面一直在调用某几个回调函数，比如 wsrep_apply_cb、wsrep_commit_cb 等，回调函数会在初始化的时候指定。所以简而言之，这个接口的作用就是用来同步数据并处理所有在集群之间转送的消息的。这些消息都是从每个节点的接收队列中提取出来，然后进行分别处理。

这个接口实际上是可以并行存在的。它对应的是参数 wsrep_slave_threads，有多少个线程，就有多少个 galera_recv 的调用，其实可以理解 galera_recv 就是一个线程函数，都在共同处理接收队列的消息，从而可以实现高性能的并行复制，保证零（基本上）延迟的特性。

galera_pre_commit

```
wsrep_status_t (*pre_commit)(wsrep_t* wsrep,
                             wsrep_conn_id_t conn_id,
                             wsrep_ws_handle_t* ws_handle,
                             uint32_t flags,
                             wsrep_trx_meta_t* meta);
```

galera_pre_commit 是 Galera 中最重要的接口之一。它的作用包括两部分，首先是将当前指定的事务写集广播给整个集群节点（自己也会收到，这一点很重要）。然后就是做验证。如果验证成功了，则将处理权交给上层，继续做数据库事务的提交操作。这个接口是在数据库事务提交时调用的，调用这个接口时，必须是本地事务已经执行完成，在提交阶段调用，只要执行到这个函数，则说明这个事务所需要的锁资源等都已经获得了。因为调用这个函数之前，数据库已经在提交阶段了，所以这里需要先调用一个接口——galera_append_data，将这个事务对应的所有 Binlog 加入到这个事务写集中。而在这之前，也就是执行过程中，每次产生对一行数据修改的 Binlog 时，都会调用另一个接口函数——galera_append_key，这与 galera_append_data 是对应的，只是调用阶段不同。galera_append_key 是将每个受影响的行，再加上这个行所在的数据库名及表名，合起来算是一个写集 Key（库名、表名、关键字字段），并加入到这个写集中。如果这个事务影响了十行，那么这个写集就会有十个写集 Key，而在提交时再通过 galera_append_data 将这个事务对应的所有 Binlog 与这些写集 Key 对应起来，形成一个（Key，VALUE）对，那么这个键值对就是这个事务的写集。加入完成之后，再调用接口 galera_pre_commit。上面已经提到，调用接口 galera_pre_commit 包括两部分，第一部分是将这个写集发送出去，第二部分是验证这个写集是不是存在冲突。不过这里涉及很

复杂的状态流转、错误处理等问题，比如验证不通过该如何处理，执行错误该如何处理等。关于这些问题，如图 29.2 所示。

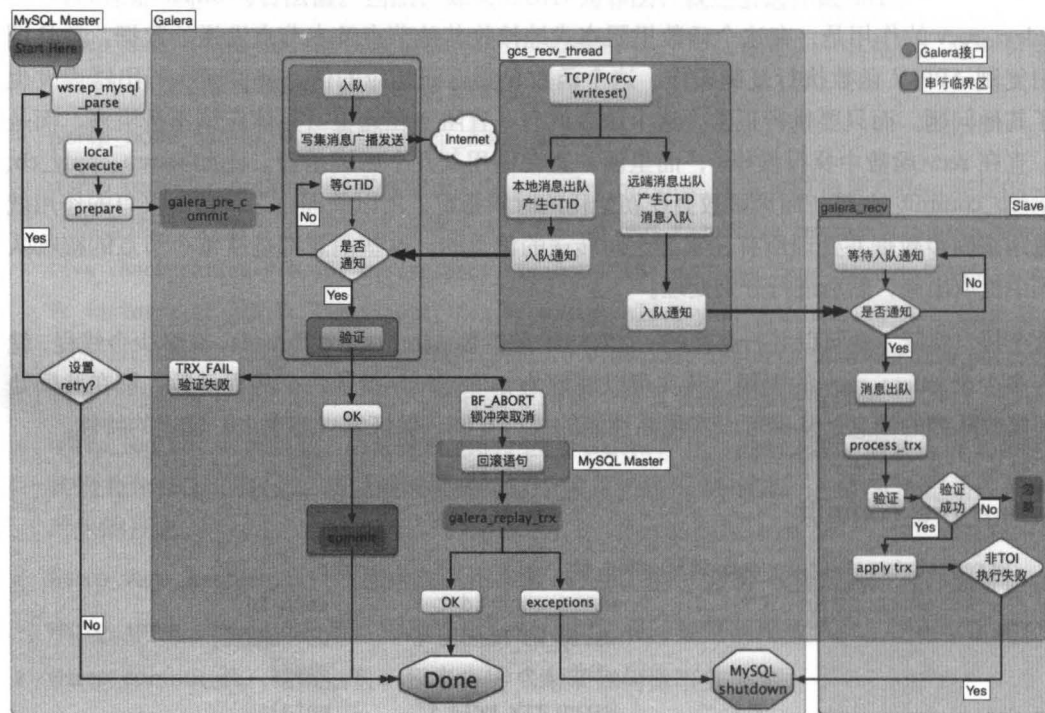


图 29.2

图 29.2 比较复杂，是接口 `galera_pre_commit` 的主要功能及状态流转。首先，`galera_pre_commit` 会将当前事务的写集加入到发送队列，然后串行（目前是串行发送的，有一个串行临界区来保护）发送出去，发送完成之后，当前事务还需要等待一个 GTID 值，等待过程是并行的，也就是说已经从临界区中出来了，这里也就是 PXC 中事务的 GTID 产生的位置。那么在什么情况下，才能等到这个值呢？还是要依赖于前面的发送，之前已经反复说过了，某个节点向集群发送一条消息，那么集群中的每个节点都会收到这个消息，而这里等待的就是本节点收到这个消息。每个节点收到消息时，都会判断消息类型，发送节点编号等，如果是本节点发送的，则本节点在接收这个消息时，会产生一个序列号，这个序列号就是对应的 GTID 值，发送到其他非本节点的消息。这里乐观地认为它们都会收到，这也是为什么一开始就说 Galera Cluster 是一种基于乐观验证的同步复制的原因。

关于 GTID 在分布式系统中，如何保证同一个事务在整个集群中顺序的问题，Galera 的解决方法是通过使用“Totem Single-ring Ordering”协议，对应的文章为 *The Totem Single-Ring Ordering and Membership Protocol*，有兴趣的同学可以下载来阅读了解。

等到 GTID 值后,就可以开始验证了,这里涉及了 Galera 的一个并发控制机制,后面会专门讲这个机制,这里只需要知道,验证是串行的(图 29.2 中的深色部分都是串行临界区域),那么这个写集就会与节点中验证缓冲区中的其他写集做冲突验证。从图 29.2 中可以看到,验证结果有三种情况,最简单的就是验证 OK,则直接将流转权交给上层 MySQL,继续做后续的提交工作即可。

如果验证时,发现有锁冲突,这里的锁冲突指的不是验证冲突,而是本节点与其他节点写入的写集存在数据库逻辑上的冲突,比如行锁。那么,此时本地事务或 Galera GTID 值大的事务(后产生的事务),会被动地被其他事务调用另一个接口 `wsrep_abort_pre_commit`,来取消当前这个事务的 `pre_commit` 操作。那么,如果此时这个事务正好在验证,在验证过程中发现状态变为被取消的状态,就会出现这种情况。MySQL 状态参数中与之对应的是 `wsrep_local_bf_aborts`,此时验证中止,并且事务回滚,但很明显这里有一个问题,验证是在后面,而前面已经将写集发送出去了,这不是数据不一致了吗?别急,Galera 还有大招,那就是再调用接口 `galera_replay_trx`,因为当前事务已经算是提交了,所有的写集都已经产生了,GTID 也产生了,这个接口此时就装做一个从节点,拿着这些信息,做一次 APPLY 操作。简单地说,就是避其锋芒了,躲避了之前产生的冲突,通过做一次 APPLY 来实现数据的执行,因为写集之前已经发送出去了,至于是不是成功,Galera 会乐观地认为此时其他节点都已经成功执行完了,所以本节点必须要再做一次。此时可能还会有读者问,如果 `galera_replay_trx` 又执行出错了呢?那只能说,这回真没办法了,Galera 目前采取的办法是自杀,也就是自己主动退出集群。

而如果 `galera_replay_trx` 执行成功了呢,那太好了,成功就成功了。这个事务就算完成了。

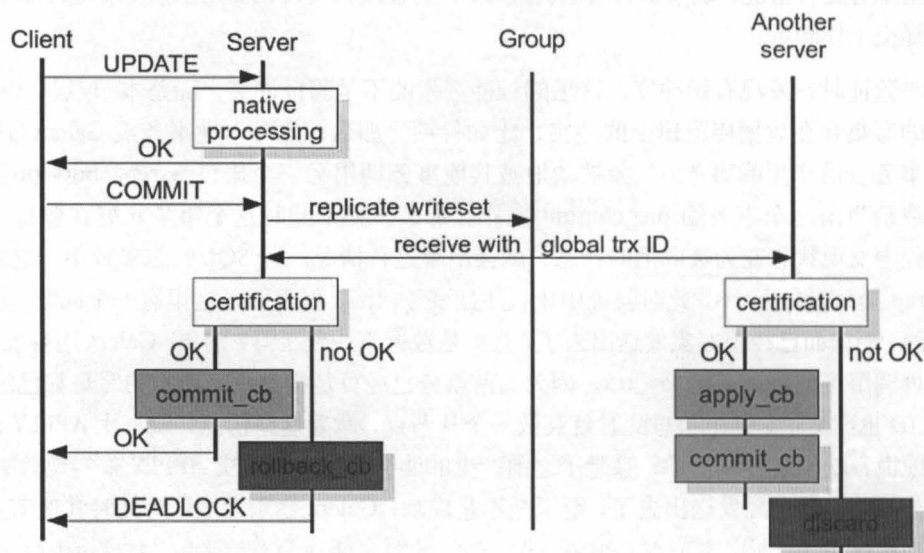
这里继续上面讲验证的三种结果,已经讲了两种,剩下的就是验证失败。何谓验证失败,第 31 章会具体讲述,这里只说结论,如果满足下面三个条件,就算验证是失败的。

- 两个事务来源于不同的节点。
- 两个事务包含相同的 Key。
- 已经存在的事务的 GTID 值,对于新验证的事务的 GTID 值,是不可见的,也就是说老的 GTID 还没有提交。

如果满足这三点,则验证就认为失败了。那验证失败的话,会如何处理呢?这里需要关注另一个参数, `wsrep_retry_autocommit`,如果这个参数设置为大于 0,则事务在回滚之后,从图 29.2 中可以看到,会再次返回去从 `wsrep_mysql_parse` 重新执行,会重试 `wsrep_retry_autocommit` 次。而这里可能又有些同学会问,还是同样的问题,当前写集已经广播出去了,本地重试的话,岂不是又会向其他节点发送重试时产生的写集么?其实是会发送,但此时的问题是如果本地验证失败了,其他节点肯定也会验证失败,那么即使是其他节点收到了,也不会执行成功,因为在同一时刻,每个节点上面的验证集合都是相同的。那么这种情况下,再次重试执

行, 是没有问题的, 在重试次数超过 `wsrep_retry_autocommit` 之后, 如果还有问题, 错误就会以 MySQL 异常的形式返回给客户端。

Galera Cluster 官网给出的一个简单点图, 主要是想讲述基于验证的复制过程, 如图 29.3 所示。



Certification Based Replication

图 29.3

这里主要讲述的是, 在正常流程下, 从事务发起到本地执行, 到 `pre_commit`, 再到验证及提交或者回滚的一个时间序列过程。从图 29.3 中可以明确地看到, 主执行线程将写集发送到 Group (集群), 然后等待 GTID 的值, 以及最终调用 `commit/rollback` 回调函数, 而从节点则调用了 `apply/commit` 回调函数。官方的图和作者画的图表达的意思是相同的, 虽然在官方图中丢失了一些细节的信息, 但是都很清楚地表达了 Galera Cluster 的工作原理。

上面所有讲述的内容, 就是 Galera Cluster 中接口 `galera_pre_commit` 所做的事情, 涉及的内容很多, 逻辑也很复杂, 同时涉及了并发控制等, 所以这个接口应该是 Galera 最重要的接口之一。

galera_replay_trx



```
wsrep_status_t (*replay_trx)(wsrep_t* wsrep,
                             wsrep_ws_handle_t* ws_handle,
                             void* trx_ctx);
```

这个接口的作用及使用时机，已经在上面的接口 `galera_pre_commit` 中讲述完了。简单来讲，就是在验证过程中，由于数据库锁的冲突，当前操作被其他线程执行了 `galera_abort_pre_commit`，导致当前线程被强制中止，但是由于写集已经复制到其他节点，所以本节点这个事务必须要完成。通过这个接口，将这个事务的写集做一次 `APPLY`，所以就叫 `replay`。

galera_append_key

```
wsrep_status_t (*append_key)(wsrep_t*          wsrep,
                             wsrep_ws_handle_t* ws_handle,
                             const wsrep_key_t* keys,
                             size_t             count,
                             enum wsrep_key_type type,
                             wsrep_bool_t      copy);
```

在接口 `galera_pre_commit` 中已经讲到，所谓的 Galera 验证，被验证的对象实际上就是写集，而构成写集的内容，其实就是通过这个接口来完成的。一个事务如果对数据库产生了修改，那么修改（删除、插入）的对象肯定是数据库表中的某一行（`Binlog_format` 为 `ROW` 模式），这一行所对应的库名、表名、主键一起组成了针对这一行数据的写集。那么每影响一行，就调用一次这个接口，如果影响多行，就会产生多组这样的信息。也就是说，如果一个事务很多，或者说影响很多行，则这个事务的写集会非常大，也正是由于这个原因，才会有两个参数 `wsrep_max_ws_rows` 及 `wsrep_max_ws_size`，可以通过这两个参数来控制一个事务的大小。如此，在 Galera Cluster 中使用的事务才不宜过大，不然会导致整个集群卡死（并发控制的问题，第 36 章会详细讲述），同时这部分的写集是用来做验证的，如果太多，验证会非常慢。当然，这只是写集的一部分内容，一个完整的写集还需要包括下面接口生成的数据。


galera_append_data

```
wsrep_status_t (*append_data)(wsrep_t*          wsrep,
                              wsrep_ws_handle_t* ws_handle,
                              const struct wsrep_buf* data,
                              size_t             count,
                              enum wsrep_data_type type,
                              wsrep_bool_t      copy);
```

上面的接口 `galera_append_key` 已经提到，Galera Cluster 复制数据，是通过写集来实现的。而写集包括两部分，一部分是用来做验证的 Keys（接口 `galera_append_key` 生成的内容），而另一部分就是接口 `galera_append_data` 所生成的内容。很多人也提出疑问，Galera Cluster 是通过什么来做到数据一致的呢？是 SQL 语句？还是 Binlog？更甚者，问到是不是通过 REDO 来做的？正确答案是 Binlog（第 31 章会专门讲解），上面的接口 `galera_append_key` 生成的是事务影响行的关键字信息，用来验证，那么很自然地想到，接口 `galera_append_data` 应该是当

前事务所生成的 Binlog 内容了。也就是说, Key 在验证通过之后, 使用 DATA 在从节点执行, 即可做到数据同步, 而这也正是一开始讲到的回调函数 `wsrep_apply_cb` 的作用, 因为 Galera 是不知道如何解析并执行 Binlog 的, 所以还需要上层来给它提供帮助。回调函数就是做这个事情的。


galera_post_commit

```
 wsrep_status_t (*post_commit)(wsrep_t* wsrep,
                                wsrep_ws_handle_t* ws_handle);
```

上面这个接口, 是用来真正提交一个事务的。而实际上, 这个事务在数据库层面已经提交, 而在 Galera 层面, 是通过这个接口来标记事务提交的。这个接口主要包括如下四个功能。

- 并发控制中, 从 Commit 临界区中出来, 这是因为调用这个接口之后, 已经在接口 `galera_pre_commit` 中, 进入了这个临界区。在这期间, 其他事务是不能提交的, 所以这个接口需要从这个临界区中出来。
- 更新状态参数 `wsrep_last_committed` 的值, 表示当前事务已经真正提交了。
- 检查当前验证写集缓冲区是不是可以做 PURGE 操作, PURGE 的条件是新提交 1024 个 Key, 或者是新产生 128M 的写集 (Gcache 占用空间大小), 又或者是新提交了 127 个事务, 如果满足则发起一次 PURGE 操作。
- 更新参数 `wsrep_local_commits` 的值, 表示本地又成功提交了一个事务。

galera_abort_pre_commit

```
 wsrep_status_t (*abort_pre_commit)(wsrep_t* wsrep,
                                     wsrep_seqno_t bf_seqno,
                                     wsrep_trx_id_t victim_trx);
```

在接口 `galera_pre_commit` 中已经说过, 如果在验证阶段, 或者是在本地事务执行阶段, 由于在 APPLY 时, 从其他节点复制过来的事务与本地事务的行锁发生了冲突, 所以 PXC 会选择一个 GTID 小的事务, 或者是本地事务作为 victim 事务。而针对 victim 事务, 根据所处的阶段不同, 采取的处理方式也不同, 如果事务已经执行过 `galera_pre_commit` 接口, 则通过调用接口 `galera_abort_pre_commit` 来取消, 否则 (victim 事务还处于本地执行阶段), 只需要将这个事务杀死, 报一个死锁的异常即可。当然, 说到 `galera_abort_pre_commit` 具体做什么事, 其实就是告诉这个 victim 事务, 当前执行已经被取消, 要尽可能地拦截下来, 而最终这个事务会被回滚。在 `galera_pre_commit` 中也说过, 这种情况, 这个节点会调用接口 `galera_replay_trx` 来 REPLAY, 这点从图 29.2 中可以清楚地看到。

galera_to_execute_start

```
wsrep_status_t (*to_execute_start)(wsrep_t*      wsrep,
                                   wsrep_conn_id_t conn_id,
                                   const wsrep_key_t* keys,
                                   size_t          keys_num,
                                   const struct wsrep_buf* action,
                                   size_t          count,
                                   wsrep_trx_meta_t* meta);
```

galera_to_execute_start 接口专门用来处理 DDL 语句的执行。DDL 在 Galera 中的执行很危险。参数 keys 和 keys_num 表示的是当前被修改对象所在的库名和表名，而 action 对应的是当前语句的 Binlog，这个 Binlog 是专门在本地构造的，而不是执行时生成的 Binlog。因为 DDL 的执行在 Galera 中是 Total Ordered 的，所以在执行时，要保证一个节点执行，其他节点也必须执行。也因此只要调用这个接口，就会直接进入 Commit 临界区，然后开始在每个节点上各自执行 DDL 语句，而 Commit 是串行的（假设参数 repl.commit_order 为 3），所以只有等这个 DDL 语句执行完成之后，其他事务才能提交。也就是说，在这个时间内，其他事务是被卡死的，卡死时间等于 DDL 语句执行的时间；或者说，在调用下面的接口 galera_to_execute_end 之后，卡死才会解除，其实这还是源于 Galera 的并发控制机制，与在 Galera Cluster 中执行一个大事务是一样的道理，只要有大的操作占着 Commit 临界区，其他事务就不能提交任何东西。这也算是 Galera Cluster 中，DDL 的一个最大的坑。

除此之外，还有 2 个坑。


- 挡我者死：由于 Galera Cluster 在执行 DDL 时，是 Total Ordered 的，所以必须要保证每个节点都是同时执行的。当然对于不是 DDL 的，也是 Total Order 的。因为每一个事务都具有同一个 GTID 值，DDL 也不例外，而 DDL 涉及的是表锁、MDL 锁（Meta Data Lock），只要在执行过程中，遇到了 MDL 锁的冲突，那么在所有情况下都是 DDL 优先，将所有使用到这个对象的事务统统杀死。不管是读事务，还是写事务，被杀的事务都会报出死锁的异常，所以这也是 Galera Cluster 中，关于 DDL 的一个闻名遐迩的坑。
- 不死之身：继上面的“挡我者死”。如果集群真的被一个 DDL 卡死了，导致整个集群都动不了，所有的写请求都夯住了，那么可能会有人想到一个妙招，就是赶紧杀死该 DDL 执行线程，直接在每个节点上输入 kill connection_id 等类似的操作。那么，此时很不愿意看到的信息报了出来：You are not owner of thread connection_id。可能有些同学要哭了。不过这种情况下，确实没有什么好的解决方法（其实这个时候，一个故障已经发生了，一年的 KPI 也许已经没有了，就看敢不敢下狠手了），要不就等 DDL 执行完成（所有这个数据库上面的业务都处于不可服务状态），要不就将数据库直接 kill 掉，快速重启，赶紧恢复一个节点提交线上服务，然后再考虑集群其他节点的数据增量的同步等。这个坑非常大，也是 Galera Cluster 中最大的一个坑，需要非常小心，避免出现这样的问题。

而说到避免这样的问题,有没有办法呢?还是有的,使用 `pt-online-schema-change` 工具便能完美解决。只是即使在这种情况下,还是避免不了“挡我者死”中,杀掉相关事务的问题。不过,这个问题一般不大,可以忍受。

所以,在 Galera Cluster 中执行 DDL 时,一定要通过 Inception 做非常严格的审核,看看影响行数有多大,可以通过参数来设置。当一个表达到一个大小,让 Inception 自动选择使用 OSC,而没有达到这个大小,Inception 会选择直接执行,而该参数 (`inception_osc_min_table_size`) 应该设置多大,需要自己把握,只要执行 DDL 的时间不影响业务即可。

很多同学都会被上面的几个问题吓倒,因此将 Galera Cluster 弃之不用。个人认为不至于,这几个坑其实都是可以简单避免的,做到知其然,知其所以然,所有问题便尽在掌握之中,所谓的问题也就不是问题了。

galera_to_execute_end

```
 wsrep_status_t (*to_execute_end)(wsrep_t* wsrep, wsrep_conn_id_t conn_id);
```

`galera_to_execute_end` 接口实际上和 `galera_post_commit` 功能一样,成对出现,是为处理不同语句而设置的,主要就是为了从 Commit 临界区中出来,从而让其他事务继续提交。

总结

Galera Cluster 目前主要是通过上面所述的接口,来实现上层与下层的逻辑、物理的数据同步功能,实现了高一致性、高可用性的 MySQL 集群。经过大量的业务应用,以及长时间的线上实践,证明了 Galera Cluster 是一个非常靠谱的 MySQL 集群解决方案。

30

Galera 参数解析

Galera 有丰富的状态参数和变量参数。通过状态参数，可以很方便地查看 Galera 引擎内部的状态，这是我们定位问题或定位报警异常的好帮手；通过变量参数，可以改变 Galera 引擎的行为，根据自己的实际使用情况按需求优化 Galera，以便满足业务需要。为了更好地帮助用户深入了解 Galera，下面会逐个讲解各种变量和参数，这对正确地使用 Galera 非常重要。

状态参数

- `wsrep_last_committed`: 这个参数表示的是当前节点最新提交的事务号，也是最新 Galera GTID 的后半部分，前半部分是参数 `wsrep_local_state_uuid` 的值。在每次执行提交、DDL 执行完成或 `APPLY` 之后，都会更新这个值。
- `wsrep_replicated`: 表示当前节点已经复制过的事务数目。这是具有相对性的。如果一直是单点写入的话，只有写节点才是有值的，其他节点都为 0；而如果是每个节点都有写入的话，那就是本节点的写入事务数。总而言之，这个参数只代表当前节点的写入数目，单位为事务个数。
- `wsrep_replicated_bytes`: 与参数 `wsrep_replicated` 相对应，每一个事务的大小不同，这个参数表示已经复制的 `wsrep_replicated` 个事务总字节大小（Key 与 DATA）的总和，原理与 `wsrep_replicated` 相同。
- `wsrep_repl_keys`: 当前节点已经复制的 `wsrep_replicated` 个事务对应的总的 Key 的数目，一个事务可以包含多个 Key，从 `wsrep_replicated` 的值与当前参数的比例上可以看出业务

特点，wsrep_repl_keys/wsrep_replicated 的值很大时，说明业务一般会在一个事务中更新很多条记录。

- wsrep_repl_keys_bytes：与参数 wsrep_repl_keys 对应，所有发送的 Key 的大小加起来的值，代表总的字节大小。
- wsrep_repl_data_bytes：与参数 wsrep_repl_keys_bytes 对应，wsrep_repl_keys_bytes 与 wsrep_repl_data_bytes 加起来的大小与 wsrep_replicated_bytes 差不多。另外，上面几个参数之间的关系是： $\text{wsrep_replicated_bytes} = \text{wsrep_repl_keys_bytes} + \text{wsrep_repl_data_bytes} + \text{wsrep_replicated} * 64$ 。
- wsrep_received：与参数 wsrep_replicated 对应，表示当前节点已经收到的从写节点复制过来的事务数，单位为事务个数。
- wsrep_received_bytes：与参数 wsrep_received 对应，表示收到的所有事务包含的 Key 及 DATA 的字节数。
- wsrep_local_commits：表示当前节点本地提交的事务个数，这个参数比较容易理解，就是每次提交之后（由接口 post_commit 来处理）加 1 就好了。
- wsrep_local_cert_failures：顾名思义，表示本地节点验证失败的次数，每次验证失败加 1，验证失败需要满足以下三个条件。
 - 两个事务来源于不同的节点。
 - 两个事务包含相同的 Key。
 - 对于满足上面两个条件的两个事务来说，相对新事务（GTID 大者）而言，老事务（GTID 小者）还是不可见的，也就是说还没有提交完成。

如果满足这三点，就认为验证失败了。

- wsrep_local_replays：所谓 replay 的意思是，当本地验证被强制取消，或者由于其他原因标志为失败之后（调用了接口 abort_pre_commit，与 pre_commit 相对应，需要注意的是这种失败与验证失败不是一个概念），会做一次本地的 REPLAY，相当于在本地重做一次。它认为既然远程节点都已经成功执行了，那么这个参数就是表示在本地做 REPLAY 的次数。这个参数值越大，表示本地出错频率越高。
- wsrep_local_send_queue：表示当前节点在等待复制的事务个数，就是发送队列的长度。虽然发送本身是串行的，但在发送完成之后，当前事务还需要等待 GTID 的产生，在这个等待过程中，还可能与其他事务提交并发送。那么，提交的事务多了，等待的也就多了，队列也就形成了。如果这个值比较大，说明本地压力或网络性能有问题，可以作为参数监控指标。
- wsrep_local_send_queue_max：当前节点历史上等待队列最大的事务数目。
- wsrep_local_send_queue_min：当前节点历史上等待队列最小的事务数目，一般就是 0。

- `wsrep_local_send_queue_avg`: 当前节点自从上次 `flush` 参数之后, 等待队列长度的平均值。其值越大则表示压力越大, 但是这个参数中没有包括 `flow_control` 的等待。
- `wsrep_local_recv_queue`: 表示当前节点从其他节点接收的队列中的事务个数, 这个队列与 Flow Control 有关, 如果这个值达到 `gcs.fc_limit` 值的话, 就会发生 Flow Control, 本节点会向整个集群发送 Flow Control 消息, 整个集群会被阻塞, 而等到 `wsrep_local_recv_queue` 的值小于 `gcs.fc_limit * gcs.fc_factor` 之后, Flow Control 解除。一般情况下, 出现这种情况是因为硬件资源不平衡, 或者从节点压力大、执行慢, 更有甚者, 硬件坏了等方面原因导致的, 可以从这方面来考虑去解决。
- `wsrep_local_recv_queue_max`: 表示当前节点历史上接收队列中事务的最大个数。
- `wsrep_local_recv_queue_min`: 表示当前节点历史上接收队列中事务的最小个数。
- `wsrep_local_recv_queue_avg`: 表示当前节点历史上接收队列中事务的平均个数, 如果集群中, 某个节点的平均值都比其他的大, 则可以考虑这个机器的硬件性能是不是不太好, 或者压力长期保持比较大的状态等。这时, 就可以考虑一下是否应该更换硬件。
- `wsrep_local_cached_downto`: 表示当前节点 GCache 中的最小 GTID 值, 可以决定集群中其他节点在启动时, 是需要做 IST 还是 SST。如果加入节点的最大 GTID (`wsrep_last_committed`) 比 `wsrep_local_cached_downto` 的值小, 则说明当前节点不能为这个节点提供数据增量服务了, 反之则可以给加入节点提供增量数据。增量范围是从加入节点的 `wsrep_last_committed` 位置开始到本节点 (donor) `wsrep_last_committed` 之间的数据增量, 也就是说, donor 节点的 `wsrep_local_cached_downto` 与 `wsrep_last_committed` 之间的差值就是整个 GCache 的容量。如果一个集群是新的, GCache 总量 (`gcache.size`) 如果还没有被写满, 则 `wsrep_local_cached_downto` 值不会变, 而在服务一段时间之后 (GCache 被写满了), `wsrep_local_cached_downto` 值会不断上涨。
- `wsrep_flow_control_paused_ns`: 表示由于 Flow Control 消息引起的集群阻塞时间长度, 单位是纳秒。这个参数不能通过 `flush status` 来重置。在 Galera 监控时, 可以通过监控这个参数来观察集群状态, 如果硬件性能不平衡, 或者某些节点压力大, 导致 FC 出现, 那么这个参数就会发生变化, 它的值是递增累计的。
- `wsrep_flow_control_paused`: 表示从上次 `flush status` 之后开始, 新产生的 FC 暂停时间与从 Flush 开始到 `show status like "wsrep%"` 命令执行的这段时间的比值, 可以理解为一个平均的暂停时间。这个值越接近 0, 说明系统越正常, 如果差不多为 1, 则说明当前系统基本不能做复制了。
- `wsrep_flow_control_sent`: 当一个节点的复制任务队列长度超过 `fc_limit` 时, 这个节点就会给整个集群发送 FC 消息, 这个参数表示当前节点向整个集群发送 FC 消息的次数。这个值越大, 表示这个节点做得越慢, 或者说如果这个值突增了, 则说明这个节点有可能出现了问题, 可能是硬件有问题, 或者是这个节点的负载增大了, 导致 APPLY 写集变慢。

- `wsrep_flow_control_recv`: 当一个从节点（相对而言）发送 FC 消息之后，其他节点就会收到这个 FC 消息（节点自身也会收到，所以有时候看监控时，`wsrep_flow_control_sent` 突增的节点，`wsrep_flow_control_recv` 也突增了，此时知道这个现象就可以了，需要重点关注的是谁发送的这个 Flow Control），那么这个状态参数就表示当前节点收到 FC 消息的次数。当这个参数突增时，就需要查看是哪个节点的 `sent` 值突增了，那么这个节点就有可能存在性能问题。
- `wsrep_cert_deps_distance`: 表示的是，一个事务的 GTID2 和它所依赖的事务的 GTID1 之间差值的总和，与这段时间内所做的总的验证通过的事务（`n_certified`）个数的比值。表示的意义是，在一个事务的等待过程中（GTID2-GTID1）其他线程做了 `n_certified` 个验证，也可以说是事务依赖的平均距离，那么这个距离的大小就可以理解为可以有这么多的事务并行 APPLY，从而可以表示并行执行的事务数目。一般可以根据这个值来设置 `wsrep_slave_threads` 的值。作为一个参照值，有时候如果 `wsrep_slave_threads` 值不够大，也会出现 Flow Control，可以通过参照 `wsrep_cert_deps_distance` 的值来设置 `wsrep_slave_threads` 的值来解决。不过通常情况下，这种情况下出现 Flow Control 是因为主库（相对）写入的 QPS（并发度）比较大，但冲突比较小，而在从库这边并发度比较小，所以导致 FC。这种问题可以通过提高并发度来解决，也就是调整 `wsrep_slave_threads` 的值。
- `wsrep_commit_oooe`: 在事务提交时，一定要保证顺序性，必须要保证 GTID 值小的事务在 GTID 值大的事务之前提交。但是经常会出现的现象是，在某一个事务（TRX1）提交时，目前最后一个提交事务（TRX2）的 GTID 比 TRX1 的 GTID 小很多（而不是小 1，说明不是连续的），也就是说在 TRX1 和 TRX2 之间，有很多事务还没有提交，TRX1 就需要提交了，那么此时只能等待这些事务直到最新提交事务的 GTID 值比 TRX1 值小 1 为止。而在并发情况下，每次出现这种等待情况时，Galera 都会累计加 1，连续加的值除以进入检查次数的比值，也就是一个乱序比例值。`oooe` 可以解释为 *out-of-order enter*。乱序不是问题，只不过是说明了事务大小不同，当前节点的压力比较大，互相影响比较大而已。但对于 `commit`，能进入提交阶段的只有一个，因为是串行的（可以通过参数 `repl.commit_order` 来修改，但一般不建议这样做，因为这样会导致集群节点产生的 Binlog 顺序不一致），所以这个值永远是 0。
- `wsrep_commit_ool`: 相对上面的参数 `wsrep_commit_oooe`，在事务提交之后，也就是离开临界区之后，正常情况下最后一个提交事务的 GTID 应该是当前事务，但发现最后一个 GTID 已经比当前事务的 GTID 大了，说明此时顺序乱了，那么这种情况出现一次，`ool` 值就加 1。这个参数的计算是用 `ool` 除以进入检查的次数，得到的比值即该参数值，`ool` 可以理解为 *out-of-order leave*。同理，对于 `commit`，能进入提交阶段的事务只有一个，因为是串行的（同上所述），从进去到出来，这些值是不会改变的，所以这个值永远为 0。

- `wsrep_commit_window`: 这个参数与上面的参数还是相关的。在每次提交时，都会检查 `oobe`，同时，也会计算最后提交事务的 GTID 与最新要进入提交的 GTID 的距离，也就是两个 GTID 之间相差的值，然后使用这个值除以进入次数的比值就得到了当前的参数值。同样的道理，如果这个值越大，则说明整个提交过程的顺序越乱，写入事务压力不太平衡；如果越接近 1，则说明写入越有秩序，事务相对比较均匀。
- `wsrep_apply_oobe`: 这个参数和 `wsrep_commit_oobe` 的道理是一样的，只是 APPLY 可以并行执行，那么这个参数就说明了在执行时，不同事务的顺序程度。如果这个值很接近 0，就说明这个系统的执行基本是串行的。
- `wsrep_apply_oool`: 与参数 `wsrep_apply_oobe` 对应，在事务 APPLY 并离开临界区之后，本来最后一个 APPLY 的 GTID 应该是当前事务的，但是却发现最后一个 GTID 已经比当前事务的 GTID 大了，这说明此时的顺序是乱的（并发执行），那么出现一次，`oool` 这个值就加 1，这个参数的计算是用 `oool` 除以进入检查的次数的比值，`oool` 可以理解为 out-of-order leave。这个参数表示的含义是，如果值越大，则表示并行执行时乱序的现象越多；如果值越小，则说明基本是顺序执行。
- `wsrep_apply_window`: 这个参数值与上面的对应，这个值越大，表示并行 APPLY 事务间的 GTID 相差越大，这个节点的活动也越频繁。
- `wsrep_local_state`: 表示当前节点的状态有如下 4 个值。
 - 1 (Joining): 表示正在请求加入集群，速度很快，一般看不到这个状态。
 - 2 (Donor/Desynced): 表示正在同步数据，当前节点为数据提供者，正在为新加入的集群同步数据。
 - 3 (Joined): 表示当前节点已经加入集群。
 - 4 (Synced): 表示当前节点与整个集群是完全同步的。
- `wsrep_local_state_comment`: 这个参数与上面参数 `wsrep_local_state` 的 4 个状态值是一一对应的，是对上面节点值的一个描述。
- `wsrep_cert_index_size`: 表示在当前节点的验证队列中，总共有多少个 Key。每次提交新的写集时，都会与这个集合来做冲突验证，并且这个集合会定期做 PURGE。与 Innodb 回滚段 PURGE 的原理类似，就是每提交 1024 个 Key、128MB 的写集（占用 GCache 的空间大小）或 127 个事务，都会做一次 PURGE，方法是找到最大的没有被依赖的事务 GTID，然后将验证缓存集群 PURGE 到这个点。通过不断观察 `show status like "wsrep%"` 的结果可以看到，`wsrep_cert_index_size` 值会不断地变化，这是因为节点需要不断地做 PURGE 操作，从而使得缓存集合不会有太多的失效事务，才能尽可能地提高验证的性能。
- `wsrep_causal_reads`: 这既是一个状态参数，也是一个变量参数。在某个节点的变量参数 `wsrep_causal_reads` 设置为 ON 的时候，在这个节点中处理读请求时，都会等待写集 APPLY 完成之后才返回结果集，这样就不会存在复制延迟了。但是，其缺点是会导致读

响应时间变长。而状态参数 `wsrep_causal_reads` 表示的是处理这种写集等待的次数, 不过这个参数已经废弃了。

- `wsrep_cert_interval`: 表示的是, 所有事务的 GTID 值与它们各自可以看到的最新提交事务的 GTID 值之间差值的总和, 与这段时间内所做的被验证通过事务的总个数的比值。简单来说, 就是当前节点中, 平均有多少个事务在验证, 或者说平均有多少个事务被复制到集群中。
- `wsrep_evsv_repl_latency`: 表示 GCOMM 在消息传送时的复制延迟, 单位为秒。看到的信息为 0.000226149/0.000557026/0.00088384/0.000268517/3, 各个值的意义分别是: 最小值、平均值、最大值、标准差、样本数量。采样时间通过参数 `evs.stats_report_period` 来控制, 默认是 PT1M (Period Time 1 Minute)。
- `wsrep_incoming_addresses`: 表示的是当前集群中, 所有已经加入或正在加入集群的节点信息, 格式为——IP: 端口, IP: 端口..., 可以通过该信息来做监控, 或者节点自动发现等一些周边工具。
- `wsrep_cluster_size`: 表示当前集群中节点的个数, 与参数 `wsrep_incoming_addresses` 对应, 也可以作为监控项, 一般监控条件必须大于或等于 3, 如果是 2 的话, 则会发生脑裂的问题。
- `wsrep_local_bf_aborts`: 表示当前节点在运行过程中, 由于事务的冲突, 导致本地事务被主动取消 (`abort_pre_commit`) 的事务个数。如果这个值比较大, 说明集群的写入冲突比较多, 可能需要调整写入方式, 比如切换写节点等。
- `wsrep_local_index`: 表示当前节点在集群中的编号。在集群中, 每个节点都有一个唯一的编号, 从 0 开始计数。
- `wsrep_ready`: 一个很重要的监控项, 可以知道当前节点的状态是不是可以服务, 正常情况为 ON, 如果变为 OFF, 则可能是发生了脑裂, 或者和其他节点之间的网络连不上, 又或者是 Galera 集群没有正常启动等。一般可以通过命令 `SET GLOBAL wsrep_provider_options='pc.bootstrap=yes'` 来恢复, 不过在执行这个命令之后, 需要观察整个集群的状态, 不然可能会导致这个节点在逻辑上脱离集群。这个命令的作用就是让当前节点变为 PRIMARY, 如果执行了, 则说明确定要使用这个节点来提供服务了。

变量参数

`wsrep_provider_options`

- `cert.log_conflicts`: 在 Galera 中, 提交的每一个事务都会做验证, 来找出事务与事务之间的依赖关系, 并检查是不是存在冲突问题等。如果在验证时, 发现了冲突, 则需要通过

这个参数来控制是否要将冲突记录下来，从而可以通过日志查出原因或用于做一些其他的工作，日志会被记录到 `error_log` 文件中。

- `gcache.dir`: 用来指定 GCache 文件的目录，只有在 `gcache.name` 参数指定的是相对路径（仅文件名）时，这个目录才起作用，而如果 `gcache.name` 指定的是绝对路径，则这个参数就被忽略了。如果这个参数没有指定，则会被设置为参数 `base_dir` 的值。
- `gcache.name`: 用来指定 GCache 文件的名称，名称中也可以带着路径，包括绝对路径及相对路径。如果是绝对路径文件名，则这个文件就以该绝对路径为准创建 GCache 文件；如果是相对路径，这个文件会创建在 `gcache.dir` 所指定的目录下，如果 `gcache.dir` 参数没有指定，则会被创建在 `base_dir` 目录下，而 `gcache.dir` 也就会被设置为 `base_dir` 参数的值。
- `gcache.mem_size`: 在第 33 章中讲到，GCache 可以由内存、内存映射文件及物理文件三部分来组成，这一点与配置有关系，参数 `gcache.mem_size` 控制的就是内存大小。GCache 的实际空间大小，其实是由内存和内存映射文件决定的，物理文件只有在 GCache 空间不够用的时候才会使用，并且很快就会被清除掉。这样，`mem_size` 的大小就决定了在整个 GCache 缓存中内存的占比大小，如果设置为 0，则表示不使用。不过，目前从官方文档中了解到，这个参数即将废弃，所以可能存在不可预知的问题，不建议使用。
- `gcache.page_size`: 上面讲到，GCache 包含三部分，如果 GCache 空间不够用，GCache 会新建物理文件来缓存新的写集，这个参数表示的就是新建物理文件的大小。如果一个还是不够用，则会继续新建同样大小的物理文件，而当 GCache 被 PURGE 之后，物理文件的 GCache 就会被清除，当文件中没有有效的写集内容时，文件也会被删除掉。所以只要物理文件出现，就预示着 GCache 已经不够用了。
- `gcache.size`: 上面讲到，GCache 包括三部分，而这个参数所控制的就是 GCache 中的内存映射文件的大小，这也是最常用的一种，设置大小对应的就是参数 `gcache.name` 所指的文件大小。当然，如果也设置了 `gcache.mem_size`，则由这两部分共同组成 GCache 的缓存区。经测试，如果将 GCache 缓存空间都由内存来组成，Galera Cluster 的整体性能会提高 10%，可惜如上所述，参数 `gcache.mem_size` 即将被官方废弃了，如果目前要使用，可能会有意想不到的坑，所以不建议使用。
- `gcs.fc_limit`: 现在已经了解到，Flow Control 可以通过参数 `gcs.fc_limit` 来设置灵敏度，设置太小的话，接收队列很容易达到这个值，这样就非常容易触发 FC，所以这个参数控制的就是接收队列达到多大时，触发 FC。
- `gcs.fc_factor`: 上面讲到，`gcs.fc_limit` 是用来控制接收队列达到多大时，触发 FC 的，而这个参数表示的是，接收队列在多大时，FC 会解除。这个参数是一个比例，`gcs.fc_limit × gcs.fc_factor` 的结果就是 FC 解除时的接收队列长度。如果小于等于这个长度，则 FC 解除。
- `gcs.fc_master_slave`: 这个参数其实与上面两个参数是相关的。如果设置为 `yes`，则表示当前集群的使用方式为主从模式，也就是单点写入的模式，其他节点都是从节点，这

样参数 `gcs.fc_limit` 最终生效的值就是其设置的值,不会发生变化。而如果设置为 `no` 的话,说明是多点写入模式,则 `gcs.fc_limit` 最终生效的值就会发生变化。Galera 会在其内部计算一个值出来作为新的 `fc_limit`, 计算方法是,将集群总节点数做一次平方根运算,然后乘以 `gcs.fc_limit`, 就是实际的 `gcs.fc_limit` 值,而解除时的接收队列长度则是实际的 `gcs.fc_limit` 值乘上 `gcs.fc_factor` 的值。

- `gcs.sync_donor`: 这个参数控制的是,在做 State Transfer 的过程中, Donor 是否要发送 Flow Control 消息。因为在给其他节点提供增量数据过程中,这个节点有可能会被影响导致执行接收队列变慢,进而产生 Flow Control,而如果将这个参数设置为 `no`,则表示不产生 Flow Control,这样就是一种非阻塞式的 State Transfer。
- `gmcast.listen_addr`: 这个值是用来感知其他节点的加入操作的,正在运行的节点会一直监听这个地址,而新加入节点在选择 Donor 之后会向这个地址发送消息,这样就可以建立联系了。我们已经知道,当某一个节点在加入集群时需要通过设置参数 `wsrep_cluster_address` 来指定连接的 IP 和端口,如果指定多个,则本地节点会选择合适的 IP 和端口加入集群。这里的 IP 和端口所指定的就是对应的已经存在的节点地址,也就是这些节点的 `gmcast.listen_addr` 参数所指定的地址。也就是说, `wsrep_cluster_address` 参数中所指的 IP 端口,要与每一个对应节点的 `gmcast.listen_addr` 参数对应,这样才能正常建立集群。比如 `wsrep_cluster_address` 指定的是 `gcomm://127.0.0.1:4106,127.0.0.2:4107,127.0.0.3:4108`,那么在机器 127.0.0.1 上面的参数 `gmcast.listen_addr` 需要指定的值就是 `tcp://0.0.0.0:4106`,在 127.0.0.2 机器上对应的是 `tcp://0.0.0.0:4107`,依此类推即可。
- `gmcast.segment`: 这个值的设置是用来在选择 Donor 时,确定哪个节点具有优先权的,范围是 0~255,值越大,优先权越高。一般在集群中机器硬件资源不一致的情况下使用,但在 Galera Cluster 中最好避免这种情况出现,所以这个参数的实际意义就不大了,一般都设置为 0,也就是自动选择 Donor。
- `ist.recv_addr`: 这个地址是用来设置做 IST 时 Joiner 的接收地址的。默认情况下,设置为 `wsrep_node_address` 所指定的地址,端口为单独的接收端口,格式为 IP:PORT。
- `pc.weight`: 这个参数是用来在 Galera 内部做 primary components 决策选举的。如果每一个节点的这个参数值都是一样的(都是 1),则选举最简单,就是可连通节点的总个数,大于原来集群节点总数的二分之一时,这些节点就是 primary components 状态的,而其他节点就变为不可服务状态,相应的 `wsrep_ready` 状态参数就会变为 OFF,这是最简单的情况。而如果设置为非 1,则选举就是按照这个参数的值来计算的,也就是将可连通节点的所有 `pc.weight` 值加起来,如果大于原集群所有节点 `pc.weight` 的总和的二分之一,则这些节点组成的集群就是 primary components 状态的,其他节点 `wsrep_ready` 状态参数值就是 OFF,即不可服务的状态。简而言之,这个参数表示的就是当前节点的权重。这个参数的目的,就是为了尽可能地避免整个集群出现不可服务的状态,但实际上,这

种情况是不可避免的，只能不断地根据不同节点的状况去优化，下面列举一些例子用来说明（摘自官方文档）。

- Weighted quorum for three nodes:

```
n1: weight 2
n2: weight 1
n3: weight 0
```

Killing nodes n2 and n3 simultaneously preserves primary component on n1. Killing n1 makes n2 and n3 become non-primary components.

- Weighted quorum for a simple master-slave scenario:

```
n1: weight 1
n2: weight 0
```

If master n1 dies, n2 will end up become a non-primary component. However, if n2 dies, n1 will continue as the primary component. If the network connection between n1 and n2 fails, n1 will continue as the primary component and n2 will become a non-primary component.

- Weighted quorum for a master and multiple slaves scenario:

```
n1: weight 1
n2: weight 0
n3: weight 0
...
nn: weight 0
```

If n1 dies, all remaining nodes end up as non-primary components. If any other node dies, the primary component is preserved. In the case of network partitioning, n1 will always remain as a primary component.

- Weighted quorum for a primary and secondary site scenario:

```
n1: weight 2
n2: weight 2
n3: weight 1
n4: weight 1
```

Site 1 has nodes n1 and n2, site 2 has nodes n3 and n4. Setting node weights as above guarantees that nodes at site 1 remain the primary component if site 2 goes down or if the network between the sites fails. Also, either n1 or n2 can crash without the rest Of the nodes becoming non-primary components.

- `repl.commit_order`: 这个参数控制的是 Galera 并发控制的行为，针对的是提交操作。为了使所有节点产生的 Binlog 完全一样，建议这个值设置为 3。在 36 章有专门的介绍，这里不做过多叙述。
- `repl.max_ws_size`: 这个参数，是用来控制写集复制大小的，单位为字节。我们已经知道，复制的内容是事务影响的 Keys 加上这个事务对应的所有 Binlog，就叫一个事务的写集，

那么这个参数所指的就是一个事务所允许的最大写集，最大可以设置为 4294901759 字节。如果一个事务产生的写集大于 `repl.max_ws_size`，则会报错，报错内容为“Got error 5 during commit”。需要注意的是，在主从到 Galera Cluster 迁移的过程中，由于主从没有这个限制，虽然有些大事务执行没有问题，但复制到 Galera Cluster 就会导致复制中断，报出的错误就是这个。此时，可能需要将这个参数调大，而正常集群写入时，建议将这个参数设置得小一些，因为现在已经知道，Galera Cluster 集群的并发控制很多都是串行的，如果事务大的话，都会影响性能，所以将此参数设置小一些可以防止大事务的产生。

- `pc.bootstrap`: 这个参数可以用来将当前节点状态为不可服务状态 (Non-Primary Components) 的节点，变为 Primary Components 状态的节点。不过，使用时需要事先确认好当前集群的整体状况，因为一般在做这个操作之前，都是已经发生脑裂现象了，所以要确认好使用哪部分节点作为 Primary Components 状态的集群来继续提供服务。如果脑裂的两个“子集群（原来集群一分为二，即形成两个子集群）”都被设置为 Primary Components 状态，就有可能产生双写，导致数据出现不一致状态，所以一般将这个参数的命令设置为 `SET GLOBAL wsrep_provider_options='pc.bootstrap=yes'`；，用于快速处理故障，恢复线上服务。

wsrep_start_position

这个参数是 PXC 版本的 `mysqld` 新增的一个参数，用来在节点启动时，指定当前节点最新的 GTID 值，或者是指定当前节点开始向集群要增量数据点的位置。当前版本的 PXC，如果使用 `mysqld_safe` 来启动的话，那么这个参数会自动通过 `--wsrep-recover` 参数来获取并指定，这部分内容的细节会在第 39 章中说明。而如果不指定由这个参数来启动的话，就会自动以 SST 的方式来启动，所以操作时需要务必小心。

wsrep_slave_threads

这个参数，是用来设置 Galera Cluster 集群中，从节点（相对）执行 APPLY 时用于做并行复制的线程个数。在 Galera 的并发控制中讲到，写集的发送、验证及提交都是串行的，只有 APPLY 是可以并行的，那么这个参数就是控制这仅有的一部分并行的并行度的。很多人可能想问，这个参数应该设置为多少比较合适，其实这是有参考对象的，即状态参数 `wsrep_cert_deps_distance`，这个参数表示的就是写集平均的依赖度，可以理解为有多少个事务可以并行执行，那么这个参数就可以设置为 `wsrep_cert_deps_distance`。不过，`wsrep_cert_deps_distance` 的值有时候会非常大（在每个事务没有任何依赖的情况下），所以并行线程数也不能跟着无限制设置，只是一个参照对象而已，因人而异罢了。

wsrep_retry_autocommit

在 29 章中已经讲到,如果写集在验证时,出现验证失败的情况怎么办?这里指的是验证失败,而不是被人强制取消。这种情况该如何处理?在 PXC 中,目前是通过这个参数来控制事务执行行为的,如果验证失败,则每个节点应该都是验证失败的,那么每个节点都会失败,从节点在验证失败的情况下,直接忽略写集即可;而主节点则是通过参数 `wsrep_retry_autocommit` 来控制,如果不为 0,则当前事务(自动提交的情况下)会再执行一次,从头到尾重新执行,然后再次发送、验证等,因为此时的操作就有可能和其他事务错开了,验证就可以通过了。当然,也还是有可能验证失败的,具体能够重试几次由参数 `wsrep_retry_autocommit` 的值来决定。

wsrep_recover

`wsrep_recover` 这个参数,是用来找到某一个节点(处于 SHUTDOWN 状态)最新 GTID 值的,它只需要去 Innodb 的 `ibdata` 的一个固定位置,找到这个 GTID 值,然后将其输出到日志文件中,从而可以了解最新的位置及与集群的差集。具体的获取方法,请参考 39 章关于 `grastate.dat` 的叙述。

wsrep_on

这个参数用来控制当前节点的写入是不是想要复制到其他节点。因为默认情况下, Galera Cluster 要保证集群的一致性,都会将一个节点的写入,复制到其他节点。`wsrep_on` 参数的作用就是用来破坏这个一致性的,设置之后,当前节点的写入就不会被复制到其他节点上了,与 MySQL 节点的 `sql_log_bin` 参数有几分相似。这个参数一定要慎用,要在明确知道自己行为的情况下使用,不然哪一天数据库自己 shutdown 了,你还不知道为什么。因为 Galera Cluster 在发现数据不一致之后,都会从集群中主动友好地退出,方式就是 shutdown。因此,这个参数一般情况就是用来主动构造数据不一致,然后做一些简单测试的。

wsrep_max_ws_size

这个参数,用来控制 Galera Cluster 在某一个节点写入的事务大小,这里指的是 Keys 及 DATA 二者加起来 Byte 数目。因为在 Galera Cluster 中,发送、验证及提交都是串行的,如果出现大事务,就会影响集群的性能,影响业务的写入,有可能引起故障,所以可以通过这个参数来设置写集的大小,如果超过这个值,直接就会抛出异常,是一种安全性的参数。

wsrep_max_ws_rows

这个参数,用来控制 Galera Cluster 在某一个节点写入事务所影响的行数。因为在 Galera Cluster 中,发送、验证及提交都是串行的,如果出现大事务,则会影响集群的性能,影响业务的写入,有可能引起故障,所以可以通过这个参数来设置事务最大影响的行数,如果超过这个值,则直接抛出异常,是一种安全性的参数。

wsrep_desync

在 Galera Cluster 中,Flow Control 是众所周知的,可以说是臭名远扬,并且因此吓走了一大批人。不过其实它并没有那么可怕,在 37 章中,已经讲到如何解决这种问题,相信肯定是奏效的。不过,这里要再说一种解决方案,那就是通过在复制延迟的节点上设置这个参数为 ON,这样这个节点就变成了一种异步复制的模式,此时主库可以一直写入,从节点也一直 APPLY,如果接收任务队列的长度已经超过了 fc.limit 值,则这个从节点也不会发送 Flow Control 消息,整个集群的写入安然无恙,只是有可能从节点的数据不是最新的了,有一段时间的延迟,如果可以接受,那长期保持这样的状态也没什么不好的。当然,如果延迟问题没有了,或者业务优化了,相信就不会再出现任务堆积的情况了,这时可以再将这个参数值设置为 OFF,整个集群又保持一致了,就变成了真真正正的 Galera Cluster。

wsrep_cluster_address

这个参数,格式类似这样:gcomm://192.168.1.100:3306,192.168.1.101:3306,192.168.1.102:3306,每一个节点,启动时都会通过设置这个参数来找到集群中的其他节点,指定的节点中可以是已经启动的,也可以是没有启动的,Galera 会自动选择并加入。建议在集群变更时尽量保证这个参数的值与集群中实际节点一致,避免在以后变更时出现不必要的麻烦。

wsrep_OSU_method

这个参数是用来控制 DDL 执行行为的。默认情况下,它的值为 TOI,全称为 Total Order Isolation,表示在执行过程中,是全程强制有序的,并且在使用到被修改的表时,都会将其杀死,这是默认情况,并且是最简单的方法,本人也推荐使用这种方法。Galera Cluster 采用这种方式执行 DDL 语句时,有很大的坑,不过在 29 章中已经讲述了如何避免这种问题的方法,所以直接使用这种最简单的设置就好。这个参数还有另一个值,就是 RSU,不过这种方法是一个节点一个节点离线改表的,比较麻烦,还有风险,不推荐使用,也不做过多的赘述。

Galera 的验证方法

Galera Cluster 或者 Percona Xtradb Cluster (PXC) 可以以 Multi-Master 架构方式使用，一个很明显的问题是，Galera 是如何解决多 Master 节点写入数据时存在的冲突问题呢？在这里 Galera 的验证机制起了决定性的作用。何谓 Galera 验证？让我们从 Binlog 与 Galera 的关系说起。

Binlog 与 Galera 的关系

使用 PXC 时，在参数中如果没有打开 Binlog，还是可以正常启动的，并且 PXC 不同节点之间的同步也没有任何问题，这给不少同学造成一种假像——Galera 或者 PXC 同步不是通过 Binlog 来同步的，这里只能说，所见非所见。

在使用的同时，有没有发现另一种问题呢，如果 Binlog 模式设置为 statement 的话，会报出 “Only Binlog_format = 'ROW' is currently supported. Configured value: 'statement'. Please adjust your configuration” 的错误，导致启动不成功，上面看到的是与 Binlog 没有关系，而这里又报与 Binlog 有关系的错误，这不是自相矛盾么，并且使得一些同学更加迷茫了，不过至少从这里可以初步确定，Galera 应该是通过 Binlog 来同步数据的。

其实 Binlog 肯定是使用了的，有这么好的现成资源为什么不用呢？

如果 MySQL 参数设置关闭了 Binlog，则在系统启动时，Galera 会自己把它打开，然后正常复制。从而导致了上面的错觉。而如果打开 Binlog 了，并且设置为 statement 模式，则 Galera 会认为你是不是有什么特殊的原因这样设置了，所以就提醒一下你，以报错的方式退出，从

而导致启动不成功的问题。只是如报错信息所示, 当前 Galera 只支持 ROW 模式的同步方式, 所以如果打开了 Binlog, 必须要设置为 ROW, 否则就别打开。当然, 打开之后, 还有一个好处就是 Binlog 会落地, 而如果关闭了, Galera 之间同步没有问题, 只是在同步之后, Binlog 都被丢弃了, 不会落地。所以还是由用户自己决定如何设置。

验证方法

在已经确认使用 Binlog 之后, 接下来的问题是, 不同节点之间如何做验证?

在之前的章节中已经讲过了, 在一个活动事务操作过程中, 因为 Binlog 模式是 ROW, 所以在当前事务中, 针对每一行“修改”, MySQL Server 都会调用 Galera 的接口 `galera_append_key` 向当前事务中加入当前行的 Key (主键列数据) 以及库名和表名 (这三者被合称为写集 Key), 当一个事务结束的时候, 产生的写集 Key 可能会很多, 表示这些行被修改了。与 `galera_append_key` 对应的是 `galera_append_data`, 这个接口的作用是, 当这个事务结束的时候, 会调用一次这个接口, 将这个事务产生的所有的 Binlog 追加到当前 Galera 事务的 data 集合中, 那么此时 Galera 的这个事务 (与上层的 MySQL Server 的事务一一对应) 就存在一个 Key 与一个 DATA, 这个键值对就是当前事务所产生的用来验证、APPLY 的。而这个 Key 与 DATA 成对组成的集合就是 Galera 中所谓的 `write_set`, 一个事务所包含的 Key 与 DATA 就被称为这个事务的写集。

如图 31.1 所示就是写集的组成结构。

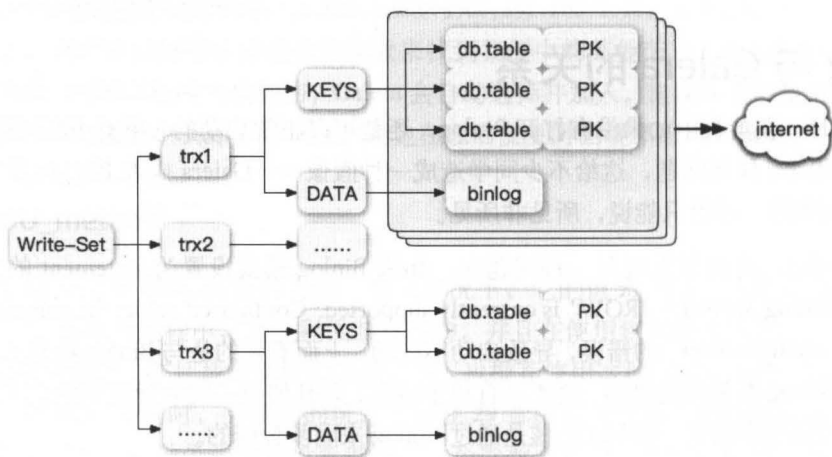


图 31.1

在当前事务提交的时候, MySQL Server 还会调用 Galera 的一个接口 `galera_pre_commit`。这个接口的作用有两个, 首先会将上面产生的 Key 与 DATA 传送 (replicate) 给其他节点, 传送

过去之后，其他节点会做验证、复制操作。这里有一个关键问题，传送过去之后，如果验证失败怎么办？复制失败怎么办？好问题，不过这里先不说了（有其他章节专门讲述这个问题）。先说这个接口在做完 `replicate` 之后，接下来就做本地的验证工作。

所谓的本地验证，就是将当前提交事务的写集与当前节点中未提交完成的事务的所有写集之间，做验证操作。当前节点的写集来源包括两部分，一部分是当前节点其他事务产生的写集，另一部分是集群中其他节点通过上面提到的 `replicate` 传送过来的写集。验证的方法是，拿着当前事务的写集 Key，与当前正在验证缓存中所有的写集 Key 对比，如果同时满足下面的三个条件，则验证不成功。

- 两个事务来源于不同的节点。
- 两个事务包含相同的 Key。
- 对于满足上面两个条件的两个事务来说，相对新事务（GTID 大者）而言，老事务（GTID 小者）还是不可见的，也就是说还没有提交完成。如果满足这三点，则认为验证失败。

本地验证完成之后，这个验证过程就算完成了，那么 MySQL Server 的两阶段提交的第一个阶段就完成了，接下来所做的就是正常的本地事务的流程了。如果验证失败，则回滚事务（有可能会重试，需要看参数 `wsrep_retry_autocommit`），如果验证成功，则提交事务。

而验证过程，就是将当前事务的写集 Keys，与当前节点的验证缓冲区中所有事务的写集 Keys 一起做对比，来判断是不是满足上面的三个条件，如图 31.2 所示。

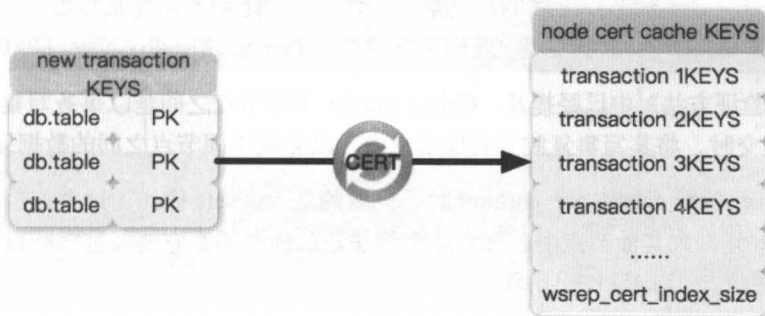


图 31.2

关于上面提到的问题，比如验证失败怎么处理，执行失败怎么处理，可以参考相关章节中的内容。

32

Galera 的消息传送

与传统的 MySQL 数据库所采用的异步复制不同，Galera 集群是以一种所谓的“virtual synchrony”来实现复制的，也可以称为虚拟同步。因为从严格意义上来讲，Galera 也不能被称为是同步的。

在这种复制模式下，不同节点之间的消息是如何传送的，比如如何复制写集，在验证失败的时候，是不是需要再次通知、确认等。我们将通过分析 Percona Xtradb Cluster (PXC) 来讲解。

在《Galera 的验证方法》中已经提及，Galera Cluster 不同节点之间是以事务写集为单位，在每一个事务提交时，将其写集复制到其他节点中，以完成不同节点之间的数据复制。

在 MySQL Server 执行 `galera_pre_commit` 时，先做的是 `replicate` 操作，这个就是将事务对应的 Keys 与 data 复制到其他节点中，那么这个传送是如何做的？它要经过哪些模块？本书将主要就 `replicate` 的传送问题作出解释。

在 Galera 中，这个 `replicate` 过程会经过多个模块，分别是 Galera 模块、GCS 模块、gcomm 模块，顺序是从上层到下层的，图 32.1 展示了整个模块之间的关系。

从图 32.1 中可以看出，整个 PXC 存在 5 层，分别是 MySQL 层、Galera 层、GCS 层、backend 层、gcomm 层，作用分别如下。

- MySQL 层：不用多讲，大家都懂得。
- Galera 层：为上面的 MySQL 层提供接口，其实就是一个框架。

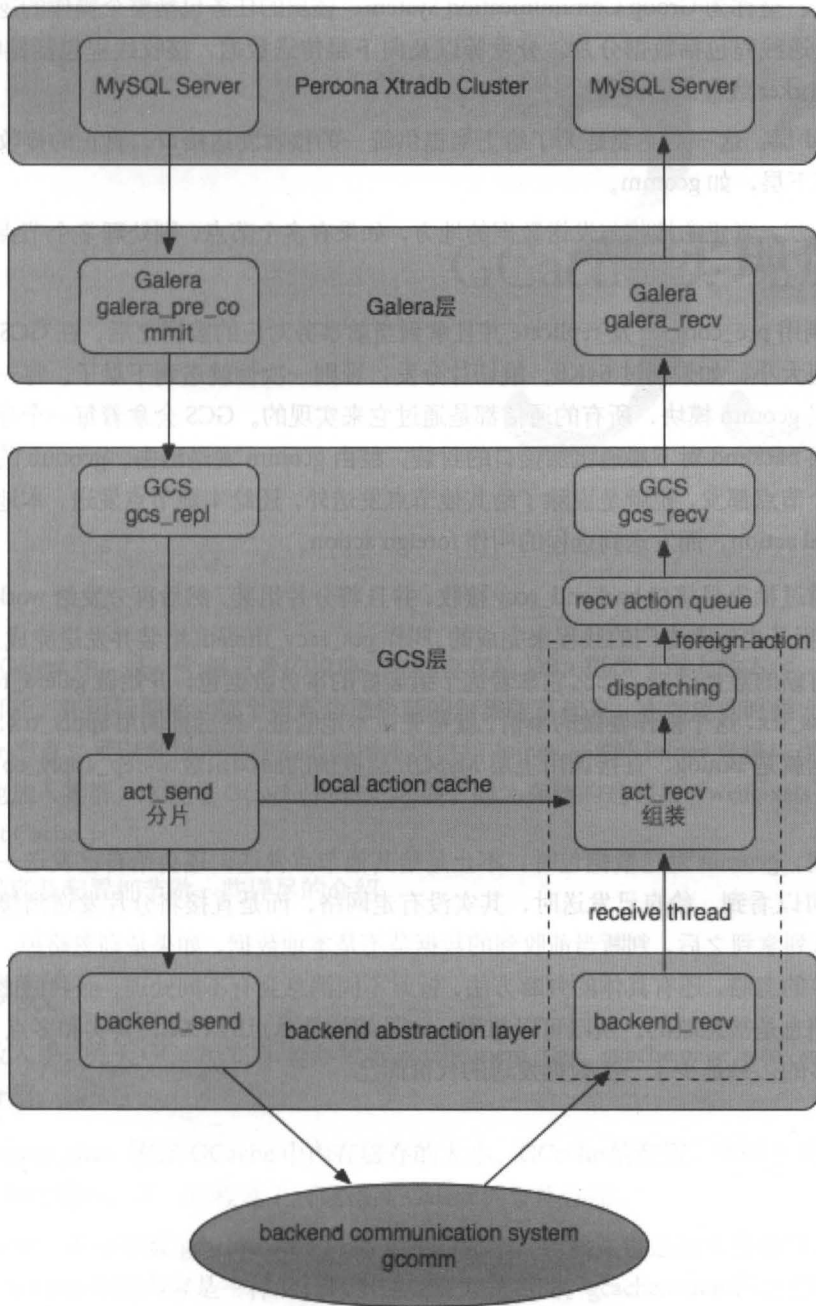


图 32.1

- GCS 层：全称为 Group Communication System，该层的任务包括整个操作的逻辑处理部分。发送线程包括数据分片、分发等以及向下层传送数据，接收线程包括接收、组装、通知 worker 线程或丢弃等。
- backend 层：这一层主要是为了给上层提供统一的接收发送接口，真正的接收发送的实现是在下层，如 gcomm。
- gcomm 层：真正的接收与发送数据的地方，如果有多个节点，则处理多个节点的发送与接收。

在 Galera 调用 `pre_commit` 及 `replicate` 并且拿到当前事务对应的数据之后，在 GCS 层判断当前事务的包大小，如果超过 64KB，就切片分发，否则一次性就送到下层了。每一个分片都会送到下层 gcomm 模块，所有的通信都是通过它来实现的。GCS 会拿着每一个分片，通过下面抽象层 backend 对下层通讯层接口的封装，经由 gcomm 发送出去，gcomm 的发送方式是给每一个节点都发，也就是说除了给其他节点发送外，还给本地节点发送，本地节点的数据叫作 local action，而发送到远程的叫作 foreign action。

远程节点通过抽象层接口 `backend_recv` 接收，并且将分片组装，然后再分发给 worker 线程处理，这个过程是由一个专门的线程来完成的，叫作 `gcs_recv_thread`。组装并发送完成之后，GCS 就会知道有新的数据过来，那么它拿着这个组装好的事务数据包，开始做 `galera_recv`，进而执行 `process_trx`，这个操作要做的事情，就是先做本地验证，然后再调用 `apply_trx`，`apply_trx` 拿到的数据就是 Binlog，直接调用上层 MySQL 层提供的回调函数 `wsrep_apply_cb` 即可完成复制。

上面还提到，gcomm 发送数据包时，不止是给其他节点发送，还会给自己发送一份，但从图 32.1 中可以看到，给自己发送时，其实没有走网络，而是直接将分片发送到接收队列中了。接收队列拿到之后，判断当前收到的数据是不是本地数据，如果是则忽略掉，但这个忽略不是简单的忽略，还有具体的判断方法，针对不同消息会有不同处理，而我们比较关心的数据复制消息是被忽略的。所以可以想到，如果架构是单点的 PXC，其实和多点 PXC 的效率是差不多的，只是少了一次数据发送的代价而已。

33

GCache 实现原理

GCache 在 Galera 中，是一个很重要的模块，它的存在给 DBA 提供了很大的方便，为简单运维提供了可能。在运行期间，每个节点会把最新的写集缓存起来，在需要的时候，如果被选择为 Donor 节点，可以直接将缓存的最新增量提供给 Joiner，这样 Joiner 直接应用增量即可简单快速地加入集群，这就是 GCache 的核心功能。因为存储的是写集 (write-sets)，所以也叫 Write-set Cache。

下面就其管理及配置细节做一些详尽的介绍。

配置参数

首先从参数入手，关于 GCache 的重要参数有 `gcache.mem_size`、`gcache.page_size`、`gcache.size`，下面分别介绍。

- `gcache.mem_size`：表示 GCache 中内存缓存的大小。GCache 的配置，实际上是可以配置一个内存区域的，在一定程度上可以提高 Galera 的整体性能。
- `gcache.size`：表示参数 `gcache.name` 所指文件的大小，当然这也是用来存储增量写集的，与内存及 `page` 存储内容是一样的，只不过实现方式不同。`gcache.name` 所指定的 GCache 缓存是通过内存映射文件来实现的，这个参数设置多少，相应的文件就会多大。不过这个参数不宜设置过大，表面上看是文件大小，但由于其实现原理为文件映射，所以当运行一段时间之后，实际占用内存大小就与文件大小相当了，会导致 MySQL 实例占用过

多内存。通过一般手段来排查问题时,很难发现其实是这个参数的设置导致无形中吃掉了太多的内存。

- `gcache.page_size`: 表示当 GCache 空间不够用,并且当前写入量还不能触发 GCache 做 checkpoint 时,会将写集保存到物理文件中,文件名类似 `gcache.page.000000`,后面的序号会像 Binlog 文件一样顺序增长。

相关参数介绍完之后,会不会有一种疑惑,一个 GCache 怎么会设置了这么多大小不一的参数?还有内存、文件映射、物理文件等,它们是如何组织的,之间又有怎样的关系?

实现原理

实际上, GCache 是有 3 层的,分别对应上面的三个参数,如图 33.1 所示。这 3 层有使用优先级,最优先使用的是 memory,如果 memory 设置为 0,或者已经用完了,再使用文件映射,如果这个也用完了,同时还没有空出更多的空间来存储新的写集,此时便可以从 page 物理文件中申请并使用。

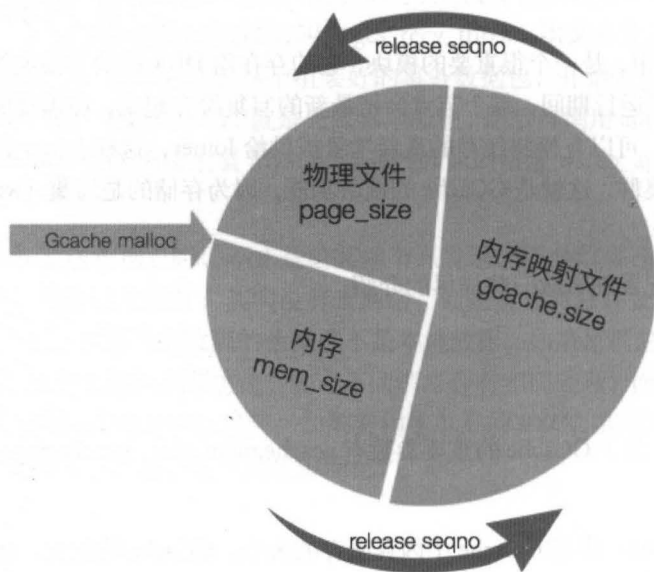


图 33.1

图 33.1 表示的即为这三者的关系,其对应的实现代码如下所示。

```

GCache::malloc (ssize_type const s)
{
    void* ptr(NULL);

```

```

if (gu_likely(s > 0))
{
    /* 计算出需要申请的空间大小*/
    size_type const size(s + sizeof(BufferHeader));
    gu::Lock lock(mtx);
    /* 先从内存中申请, 如果mem_size设置为0, 则返回值ptr为NULL, 继续从rb中申请 */
    ptr = mem.malloc(size); /* malloc from memory firstly */
    /* rb即为内存映射文件, 是RingBuffer的简称 */
    if (0 == ptr) ptr = rb.malloc(size); /* malloc from memory secondly */
    /* 如果上面的rb申请还是返回NULL, 则最后从PageStore中申请, 即为文件存储。
       这种情况下, 肯定可以申请到空间, 否则就有问题了。 */
    if (0 == ptr) ptr = ps.malloc(size); /* malloc from memory lastly */
}
return ptr;
}

```

从这段代码中可以看出 Galera 的管理方式及其组成结构。

但是, GCache 是什么时候才需要申请呢? 也就是说, 一个事务缓存到本地, 是靠什么来驱动的呢?

现在已经知道, Galera 的工作模式是——某个节点写入一个事务, 它会广播到其他节点, 而这个所谓的其他节点, 也包括自己。也就是说自己发出来的事务, 自己也会收到, 只是在收到并产生 GTID 之后, 就被简单忽略了, 而不会再去 APPLY 一次。那么这里就注意到, 不管哪个节点写入, 哪个节点接收, 其实有一个共同点, 就是每个节点都会接收, 接收到的信息会在消息组装时缓存到 GCache 中, 而此时就用到了 GCache, 而这也是在“正常运行”时唯一一处用到 GCache 的地方, 因为如果不做 IST 或 SST 的话, GCache 就没有用处了。

上面讲到了 GCache 的申请过程, 但 GCache 不可能是无限扩展的, 空间是有限的, 如果把设置的空间使用完了, 怎么办呢? Galera 此时会用到一个我们很熟悉的 PURGE 操作。和 InnoDB 事务类似, 这也是用来清除空间的, 所有已经提交的事务所对应的 GTID 值必须要被标记为可见, 这样不管是验证缓冲区, 还是 GCache 空间, 都可以清除一部分过期(已经提交)的事务, 一方面可以提高性能, 另一方面清出一些空间以便其他事务使用。

Galera 在运行过程中, 每次向 Galera Cluster 提交事务时, 都会去更新当前节点的 last_committed 值, 对应 status 参数 wsrep_last_committed, 此时会判断是否需要做 PURGE 操作, 如果当前节点处理的写集键 (Keys) 数大于 1024, 或者写集字节数大于 128MB, 又或者事务数大于 127 个, 则将这些计数清零, 并且做一次 PURGE 操作, 否则事务就提交完成了。

如果需要做 PURGE, 那么此时这个操作会激活 Galera 的一个后台线程 ServiceThd, 线程收到这个信息后, 会通过 set_last_applied 向集群发起一个协议消息, 消息类型为 GCS_MSG_LAST。这个消息, 集群中的每一个节点都会收到, 每个节点通过 gcs_core_recv 接收到之后, 再将

这个消息类型修改为 GCS_ACT_COMMIT_CUT。因为 gcs_core_recv 只负责接收 TCP/IP 消息，而处理消息是由 galera_recv 接口来处理的，所以在修改消息类型之后，再次打包将这个 message 放到接收队列中，那么接收线程的工作就算完成了。galera_recv 感知到接收队列中有新消息之后，将其取出并通过 galera::GcsActionSource::dispatch 来处理，此时处理的消息类型就是 GCS_ACT_COMMIT_CUT，具体工作如下。

- 本节点会将消息传入的 last_committed 值，与当前安全可 PURGE 的 safe_gtid（没有任何事务依赖这个 ID）相比，取最小值，计算出要消除的 gtid 的最大值 purge_gtid。
- 开始做 GCache 的消除工作。由本节点发起，将这个操作再次交给服务线程 ServiceThd，只是这次只会做本地处理，不会再向集群发送消息了。不过，此时还是要选择一个合适的量来做 PURGE，不会直接 PURGE 到 purge_gtid，因为每次 PURGE 都有一个批量处理值，默认是 32，如果上次 PURGE 到的 GTID 值 seqno_released 与 purge_gtid 相差大于批量处理数目的 2 倍，则本次处理数目为批量处理数目值（即 batch_size，参数在后面做解释），否则处理个数为 purge_gtid-seqno_released。这里的批量处理数目是会动态变化的，如果每次处理时，发现最大 GTID 与上次 PURGE 到的最小 GTID 值 seqno_released 之间的距离变大，则批量处理值会线性地每次加上 32，而如果距离变小，就不会再加 32，这一步计算出来的数量称为 batch_size。
- 从 seqno_released 位置开始 PURGE，PURGE 事务数为 batch_size，每处理一个，更新一次 seqno_released。不过本次处理不会真正释放 GCache 空间，只会将这些事务对应的状态改为 RELEASED，这些事务的缓存还是可以使用的，只是表明临近释放了。如果标记为 RELEASED，发现某一事务是存储在物理文件中的，而不是内存或内存文件映射，那么此时说明 Gcache 已经用完了，就会触发真正的释放，叫作 discard。
- discard 操作，会从 GCache 的最开始位置，或者当前节点最小的被缓存的 GTID 开始扫描，将被标记为 RELEASED 状态的缓存事务的内存及文件占用空间等释放，直到遍历到当前 seqno_released 值结束。然后继续做 PURGE，因为此时可能有 batch_size 个的 PURGE 还没有完成，但一般情况下，这批 PURGE 操作后面几个事务的 PURGE，都会做一次 discard 操作。
- PURGE 完成之后，直接的结果就是 GCache 会释放一部分出来，释放的对象是当前节点最老的事务缓存，那么相应地，当前节点可提供的增量数据也减少了，对应的状态参数为 wsrep_local_cached_downto。查看这个参数时，如果这个参数一直在变，说明 GCache 一直是满的状态，而如果长时间不变，说明 GCache 应该还有空闲空间没有使用。这也是观察一个节点或整个集群压力大小的指标。

这里还需要额外说明一点，即 seqno_released 值是从 GCache 产生，一直随着当前节点 wsrep_last_committed 值而增长的值。它是一直增长的，但总是会比 wsrep_local_cached_downto 大，并且 seqno_released 值的变化频率会比 wsrep_local_cached_downto 的变化频率大，因为

`wsrep_local_cached_downto` 只有在空间不够用并产生新的写入导致发生 `discard` 的时候才会增长，而 `seqno_released` 会一直增长。

那么同样可以看到，只要释放一点空间，这些空间就会被使用到，因为申请空间的顺序是固定的，申请时可以很快地找到这些空闲空间并且拿来使用。但需要注意的是，在配置内存与内存文件映射大小比例上面，相比纯内存的 GCache 配置方式，配置的大小比例是多少，就有多大比例的性能提升。目前根据一些测试，如果配置为全内存，大概可以提高 10% 的写入性能。

另外现在可以知道，虽然可能通过参数 `gcache.size` 指定 GCache 的大小，但真正的 GCache 的大小是 `gcache.size` 加上 `gcache.mem_size` 的值。当然，如果 `gcache.mem_size` 设置为 0 的话，`gcache.size` 也就代表其本身的意义了。

GCache 的大小设置是由公式算出来的，并不是随便设置的。因为它的本质意义是为其他节点提供增量数据，所以它缓存得越多，则提供得越多，从而在 DBA 运维的过程中，提供给其他节点用来停机运维的时间也就越长，不用担心数据库维护完成之后，需要重启做 SST，这样就不划算了。而如果 GCache 设置得太大也是有坑的，因为 GCache 使用的两种存储介质，其实最后都会落到内存上。虽然内存文件映射看上去是一个文件，但是别被蒙骗了，如果 GCache 空间用完了，实际上还是占用了相同大小的内存空间，这个问题会导致数据库吃内存非常严重，并且很难察觉到是因为这个导致的。

所以，设置一个合适的 GCache 大小是很有必要的。如何才是合适的，这就需要考虑一般需要多少运维时间了。假设需要 `optime` 小时，而每秒的流量为 `flow_per_second`（计算方法为计算出每秒钟 `wsrep_received_bytes` + `wsrep_replicated_bytes` 的增量值，单位为字节）。

计算公式为：
$$\text{gcache_size} = \text{flow_per_second} * 3600 * \text{optime} / 1024 / 1024 \text{ (G)}。$$

发散思维

有人可能想问，如果将 GCache 的两个参数：`gcache.size` 及 `gcache.mem_size` 都设置为 0 的话，那将会是什么现象呢？

现在已经明白，除了这两部分之外，还有一个是物理文件的 GCache 缓存，只有前两个缓存不够的时候，才会将新的写集存储到物理文件。

试想一个问题，如果参数设置前两部分都为 0 了，那该如何？其实结果可想而知，相当于这个节点的 GCache 都通过物理文件来存储。我们现在已经知道，当 GCache 中有物理文件存储时，Galera 每隔一段时间就会做一次 PURGE，将 GCache 空间释放出来，将参数 `wsrep_local_cached_downto` 增大。

这种情况下,会发现 `wsrep_local_cached_downto` 值变得非常快,基本上与状态参数 `wsrep_last_committed` 很接近。通过简单的统计,不难发现,其实相差的值基本上与两次 PURGE 的间隔值差不多,也就是说,基本上是每做 127 个事务或写集累计大小为 128MB,抑或是写集 Keys 累计为 1024 个时,都会做 PURGE 操作。

现在也就更深刻地明白了,其实 GCache 的物理文件中存储的数据最多就是这三者的最小值,每次达到这个值后,很快就被 PURGE 掉了。

当然这个设置是极端情况。这种情况下,性能必然很差,并且也基本不能为其他节点提供 Donor 服务了。

34

大话 SST/IST 细节

这些年，我们一直在使用并推广基于 Galera Cluster 的 Percona Xtradb Cluster (PXC)，到目前为止 PXC 已经被业界熟知了。它是一个可以保证数据一致性的高可用架构，每个节点的数据都是自动同步的。因为支持多点写入，所以不管在哪个节点上写入数据，都会自动同步到其他节点，关于数据的在线同步在第 32 章中已经讲述。本章要讲的是，当一个新节点加入的时候，PXC 或 Galera Cluster 是如何实现的，执行流程是什么，它怎么来保证数据的一致性。

当需要新增一个节点时，必须要指定参数 `wsrep_cluster_name` 为要加入集群的集群名，同时要指定 `wsrep_cluster_address` 为要加入集群的某一个在线节点的地址，或者几个地址。其他必要的配置这里就不多说了，在很多 PXC 安装配置的文章中应该都说过。

何谓 SST、IST？这是在 Galera 运维过程中一直要提及的名词，它们的实质是 ST，即 State Transfer，就是将一个节点加入集群中，从而做到让这个节点 State Transfer。有两种方式可以做到这一点，一种是基于某一个 state 做增量 (incremental) 的 transfer，这种方式被称为 IST (Incremental State Transfer)；另一种是如果还没有基础数据，那就需要通过数据快照 (snapshot) 做全量的 State Transfer 了，这种方式被称为 SST (State Snapshot Transfer)。

那么，新节点 (要加入集群的节点) 在这里要做的事情包括下面几点。

初始化节点环境

- 这里首先要做的事情是载入 Galera 层的共享包 `libgalera_smm.so`, 所有关于 Galera 接口及数据同步的实现都在这个包里面, 该共享包的相关信息, 可以在 MySQL 日志文件中找到对应的信息 ([Note] WSREP: wsrep_load(): loading provider library '/var/mysql/galera/lib/libgalera_smm.so')。
- 接下来就是初始化 Galera 用于处理上层数据逻辑的一些回调函数及一些参数, 先看一下代码实现, 如下。

```

1 wsrep_args.data_dir      = wsrep_data_home_dir;
2 wsrep_args.node_name     = (wsrep_node_name) ? wsrep_node_name : "";
3 wsrep_args.node_address  = node_addr;
4 wsrep_args.options       = (wsrep_provider_options) ? wsrep_provider_options
  : "";
5 wsrep_args.logger_cb     = wsrep_log_cb;
6 wsrep_args.view_handler_cb = wsrep_view_handler_cb;
7 wsrep_args.apply_cb      = wsrep_apply_cb;
8 wsrep_args.commit_cb     = wsrep_commit_cb;
9 wsrep_args.sst_donate_cb  = wsrep_sst_donate_cb;
10 wsrep_args.syncd_cb      = wsrep_syncd_cb;
11 rcode = wsrep->init(wsrep, &wsrep_args);

```

- 第 1 行很明显就是 MySQL 要告诉 Galera, 数据目录在什么位置。
- 第 2 行就是让 Galera 知道当前新加入的 PXC 节点的名字是什么。
- 第 3 行就是要说明新加入节点的地址是什么, 即一个 IP 地址。
- 第 4 行是 PXC 的一个设置参数, 对应的是 `wsrep_provider_options`, 这里主要设置一些 Galera 内部的控制运行方式等一些参数, 具体的参数可以参考另一篇文章。
- 第 5 行是设置一个回调函数, 每次 Galera 要记日志了, 会通过调用这个函数来实现, 因为下层不知道上层的日志逻辑及位置, 所以提供这个函数之后, Galera 就可以记录下来一些有用的信息, 这些信息可以在错误日志中找到, 那些以“WSREP:”开头的日志条目应该都是通过这个回调函数打印出来的, 比如: 2015-05-09 12:36:36 57936 [Note] WSREP: gcomm: connected。
- 第 6 行是本章要重点讲的一个回调函数, 每当一个新节点加入的时候, 都会调用这个函数。
- 第 7 行是用来同步数据的, 这个函数在从节点调用, 当写节点执行一个写入后, 从节点就会执行该函数做数据复制, 因为 Galera 层是不知道 Binlog 是什么东西的, 所以只能由上层来操作逻辑的实现。

- 第 8 行从名字就可以看出是提交操作，当一个事务完成之后，就会调用这个函数，它与上面的复制函数对应，每复制一个事务，就会做一次提交。
- 第 9 行也是这篇文章的主题，它负责处理 SST 过程中 Donor 部分。
- 第 10 行处理的是节点的 Synced 及 Desynced，这里不做过多介绍。
- 最后一行就是对上面这些设置做初始化，把这些信息告诉这个节点。

连接到集群并且做 SST/IST

当上面初始化完成之后，接下来的工作就是连接集群了。这个由 Galera Cluster 中的函数 `wsrep_start_replication` 完成，其内部通过调用 Galera 的接口 `galera_connect` 来实现。这里在执行完成之后，当前新加入的节点就会成为集群的一部分，成为集群中可见的节点，此时登录正常节点可以执行以下操作。



```
show status like "wsrep%";
```

在执行结果中，可以通过参数 `wsrep_incoming_addresses` 来查看当前集群中已经存在的节点，如果里面有想要加入的节点，则说明已经加入成功。

下面讲述一下 Galera 是如何在启动的时候，触发 SST 操作的，最后可以看到，启动 SST 的过程，其实就是消息的流转过程。

GCS_MSG_COMPONENT 消息

上面已经提到，在调用 Galera 的接口 `galera_connect` 时，实际执行的是 `gcs_open`，就是启动一个节点的第一步，而一个很重要的操作就是创建后端接收线程 `gcs_recv_thread`，这个线程的作用是处理一切节点之间消息接收及处理等操作。实际的最终网络接收处理函数为 `GCS_BACKEND_RECV_FN(gcomm_recv)`，这个函数里面有三个处理分支，最后一个分支用来处理新增节点，这个新节点的第一个消息也是从这里发出。在 Galera 中，所有的消息都是发送给整个集群的，只是不同消息的处理方式不同，当然有些还会直接丢弃。第一个消息的类型为 `GCS_MSG_COMPONENT`。实际上，它是在这个发起处给自己构造了一个消息，算是无中生有。因为这是一个比较底层的消息接收函数，所以在这里构造出一个消息来，在运行堆栈上层看起来好像是收到了这样的消息，但是在线程 `gcs_recv_thread` 的消息处理函数 `gcs_core_recv` 中会直接将其处理掉。对于类型为 `GCS_MSG_COMPONENT` 的消息会在函数 `core_handle_comp_msg` 中处理，集群中的每个节点都会处理它。首先，每个节点都会将本地集群状态都变为 `GCS_GROUP_WAIT_STATE_UUID`。然后，Galera 会选择一个代表（一般是编号为 0 的节点），生成一个临时 UUID，再发送一个 `GCS_MSG_STATE_UUID` 类型的消息给整个集群，这是要告诉其他节点当前的纪元是什么，因为此时的 UUID 就是一个可

以临时作为纪元意义的东西。而集群中的其他节点收到这个消息,并完成这一步之后,新节点已经将 UUID 发送完成,老节点都已经接收到了这个 UUID,接下来要处理的消息类型就是 GCS_MSG_STATE_UUID。

GCS_MSG_STATE_UUID 消息

通过函数 `core_handle_uuid_msg` 来处理,处理方式很简单,就是从消息中取出包含的临时 UUID 值,也就是纪元,这是为了能达成一致协议而使用的。将自己的节点状态修改为 `GCS_GROUP_WAIT_STATE_MSG`,表示下一步就是等待状态变更的消息。在上面讲的 `GCS_MSG_STATE_UUID` 消息处理返回之前,如果状态为 `GCS_GROUP_WAIT_STATE_MSG` (实际上每个节点的状态此时都是这个),则会向集群发送一个消息,消息类型为 `GCS_MSG_STATE_MSG`,这主要是想要发出与集群同步状态的信号。那么,每个节点都发送这个消息,消息发送的同时,带着当前 `GCS_MSG_STATE_UUID` 类型消息中的 UUID 值,作为消息同步的纪元值,这样相当于是每个节点都会收到集群节点个数的 UUID 状态信息。

GCS_MSG_STATE_MSG 消息

状态为 `GCS_GROUP_WAIT_STATE_MSG` 的节点会收到这个消息(每个节点都是这个状态),这个消息的内容就包括了上面收到的临时 UUID (因为这个 UUID 就是用来同步的)。

对这个消息的处理方式是,首先将每个节点发过来的这个消息中的 UUID 及 `state` 取出来,然后根据这些信息,将当前节点中对其他节点状态的缓存信息更新掉,然后再根据每个节点上面对其他节点状态的缓存做一次选举。因为此时每个节点都收到了三个相同的临时纪元 UUID,所以这里的选举是可以完成的,可以选出来集群中哪个节点是最新的,然后以这个节点为中心,来同步其他节点的数据,然后将集群状态修改为 `GCS_GROUP_PRIMARY`。同时,根据选举结果,如果当前节点的 GTID 与选举的最大 GTID 相同,则说明这个节点是 Synced 状态的,节点状态保持不变,为 `GCS_NODE_STATE_SYNCED`;而如果不相同,则说明节点与集群存在数据差,需要将这个节点的状态修改为 `GCS_NODE_STATE_PRIM`,表示 “in primary conf, needs state transfer”,此时新增节点 Joiner 和已经在集群中的节点状态就不同了,后面节点所要做的操作都会因此而产生差别。

在 `GCS_MSG_STATE_MSG` 消息处理返回之前,因为此时的状态已经被修改为 `GCS_GROUP_PRIMARY`,所以每个节点再构造一个消息类型为 `GCS_ACT_CONF` 的消息也属于无中生有。此时的处理与上面有所不同,上面的消息只需要经过 `gcs_rcv_thread` 处理,并且再次发送新的消息来实现消息的轮转,而类型为 `GCS_ACT_CONF` 的消息还需要在本地转发出去,交给接收队列来处理,也就是 `dispatch`,而这种类型的消息被称为 ACTION 消息。


GCS_ACT_CONF(ACTION)

这个 ACTION 通过函数 `process_conf_change` 来处理，因为接收到的是上面发出的配置变更信息，所以里面包含每个节点自己的 GTID 信息。每个节点在处理这个消息时，都会判断本地的 GTID 与集群同步的 GTID 是否相同，因为老节点都是同步的，所以什么都不需要做，而新节点是不同步的，所以需要继续处理这个情况。

在判断是否需要做同步的时候，其实存在两个条件：首先节点状态是上面说过的 `GCS_NODE_STATE_PRIM`；其次，在这个基础上再判断 GTID 的大小关系。

对于已经加入的正常运行的节点，调用回调函数 `wsrep_view_handler_cb` 的时候，因为状态是同步的，里面不需要做状态同步 (SST/IST)，所以只需要做一些简单的操作，比如修改 `auto_increment_offset` 及 `auto_increment_increment` 等，用来调整在节点个数改变之后自增 ID 的增量步长值。

对于新加入的节点，因为需要做 SST/IST，此时执行的还是回调函数 `wsrep_view_handler_cb`，只是行为方式不同，这个函数会根据配置文件中设置的参数 `wsrep_sst_method` 来决定如何做 SST，最终会生成下面这样一个命令。

```
 wsrep_sst_xtrabackup-v2 --role 'joiner' --address '10.88.132.213:4308'
--auth 'sstuser:password' --datadir '/var/mysql/multi/3308/data/'
--defaults-file '/var/mysql/multi/3308/etc/my.cnf' --parent '31487'
--binlog '/var/mysql/multi/3308/binlog/mysql-bin'
```

从命令中可以看出，这里选择的方式是 `xtrabackup`，所以执行的脚本是 `wsrep_sst_xtrabackup-v2`。

在 MySQL 层，会创建一个单独的 `sst_joiner_thread` 线程去执行上面的命令，具体如何执行，以及如何与 MySQL 层交互，还需要具体看 `wsrep_sst_xtrabackup-v2` 脚本的实现。

简单地说，脚本 `wsrep_sst_xtrabackup-v2` 是两用的，既处理 Joiner，也处理 Donor。在处理 Joiner 时，MySQL 层首先要等待底层接收通道已经准备好，在脚本启动之后，会输出一个“READY”，上层 MySQL 读取到之后，说明接收脚本已经准备好，脚本此时的工作就是静静地等待 100 秒，等待 Donor（现在可能还没有选举出来，但 100 秒之内应该可以找到一个合适的）发送 GTID 信息。

其实，是上面的脚本来决定要做 SST 还是 IST 的地方。如果 Donor 判断是要做 IST，则它会发送一个 IST 文件过来；如果是要做 SST，则它会发送 GTID 信息过来，而 Joiner 是通过判断这两个文件来区分应该如何接收数据的。

如果 100 秒还是没有收到，则此次 Join 就会终止。如果收到了，`sst_joiner_thread` 线程的工作就是等待 State Transfer 的结束（注意，此时还没有开始 ST，只是在等待下一步的操作）。

在等待的过程中, MySQL 启动的主线程首先会修改 MySQL 节点参数 `auto_increment_offset`、`auto_increment_increment`。然后因为需要做 SST/IST, 所以对于新加节点来说, 会先将 Joiner 节点的状态改为 `Joining`, 这个状态对应的是状态参数 `wsrep_local_state_comment`。之后在 Joiner 节点上通过函数 `request_state_transfer` 向集群发送一个 `GCS_MSG_ACTION` (ACTION 的类型为 `GCS_ACT_STATE_REQ`) 类型的消息, 这个消息的作用就是找一个 Donor 出来, 然后提供数据的同步操作, 而选择 Donor 是集群中的哪一个节点, 都由集群中的每一个节点来决定。

函数 `request_state_transfer` 首先要做的操作是确定做 SST 还是 IST, 通过调用函数 `prepare_for_IST` 来判断, 它会判断在新节点启动时就读到的文件 `grastate.dat` 中的 UUID, 以及调用 `wsrep_init` 接口时传入的当前节点的最新 GTID 值是否有效。如果在启动参数 `wsrep_start_position` 指定的 GTID 中, UUID 为 `00000000-0000-0000-0000-000000000000`, 或者与要加入的集群 `wsrep_cluster_state_uuid` 值不相同, 则直接只做 SST; 如果 UUID 相同, 并且参数 `wsrep_start_position` 指定的 `seqno` 为一个有效的值, 此时会先做 SST, 然后再做 IST 操作 (需要注意的是, 需要做 IST 的时候, 其实 SST 也是要做的。这里的 SST 就是一个标志作用, 表示的是基础数据已经完备了, 可以处理增量数据了, Galera 在等待上层数据库启动成功并调用一次接口 `galera_sst_received`, Galera 知道数据库已经准备好 APPLY 数据了, 则再继续做 IST 操作, 在这期间, Galera 一直处于等待状态)。

还有一点需要注意, 这里判断出来该做 SST 还是 IST, 是初步判断的结果, 还需要加上 Donor 的判断。只是 Joiner 单方面决定是不够的, 因为如果 Joiner 决定做 IST, 但实际在 Donor 端, Donor 的 Gcache 不够用了, 还是会转换为 SST。

其实在这期间, Joiner 节点启动主线程一直处于 SST 阶段的等待状态, 在等待 `galera_sst_received` 的调用, 等待一个标志, 即告诉 Joiner, SST 完成了。而具体到 Donor 给 Joiner 传的是全量备份数据, 还是增量数据, 全靠 Donor 如何调用回调函数 `wsrep_sst_donate_cb` 来决定。如果发送的是全量数据, 则 Joiner 这边的 `sst_joiner_thread` 线程会接收 `xtrabackup` 备份的数据, 接收完成之后, 会调用 `galera_sst_received` 接口通知启动主线程 SST 已经完成, 而如果经 Donor 判断之后, 可以做 IST 操作, `sst_joiner_thread` 线程会直接调用 `galera_sst_received` 通知线程启动。这个过程会非常快, 也就是说不管做 SST 还是 IST, SST 的过程都是需要的, 不同的是时间长短问题。

有一个问题是, 当最终的 ST 方式与 Joiner 开始的判断不同时, Joiner 该如何应对呢? 也就是说, 在 Donor 判断之后, 发现 GCache 不够用, 不能直接给 Joiner 提供增量数据, 此时 Joiner 本来按照原计划是要做 IST 的, 但接收到的是 SST 全量数据, 此时接收到的数据中, 最新的 GTID 值必然大于在做 SST 之前的集群的 GTID 值, 那么此时 IST 就会被跳过, 从而实际上是做了一个全量的 SST (一个 IST 的华丽转身)。

如果是直接做 SST, 那么在调用 `galera_sst_received` 接口之前, 需要等待 Donor 将 xtrabackup 或其他方式备份的数据传到 Joiner 端 (此时 Joiner 拿到的数据就是某一时刻的数据库镜像, 可以正常启动了), 并且数据库启动之后, 继续做加入集群剩下的工作。

下面继续讲述 Joiner 在加入时, 如何通知 Donor 来给它传数据过来, 这就要从消息 `GCS_ACT_STATE_REQ` 说起了。如果要做 IST, 则消息内容会包含如下信息: `b00b6da6-2647-11e6-8c79-db93489ea7dc:50264-50267|tcp://10.90.57.155:4206`, 格式为: `uuid:start_gtid-group_gtid|tcp://recv_addr:recv_port`。

GCS_ACT_STATE_REQ(ACTION)

处理消息的函数为 `core_handle_act_msg` (又见到了), 因为消息处理分两层, 第一层为消息接收线程, 类型为 `GCS_MSG_ACTION`, 第二层为接收队列线程, 这个队列是由第一层接收之后, 经过处理交给第二层的, 这种消息叫 ACTION 类型。

具体处理此 ACTION 的函数为 `gcs_group_handle_state_request`。在这里, Galera 要选一个 Donor 出来, 给 Joiner 提供数据, 这里分两种情况: 如果在 Joiner 启动时, 指定了 Donor, 则在消息中取 Donor 名, 直接选定这个为 Donor; 而如果没有指定, 则为空, 就需要选一个出来, 选择 Donor 的原则是找一个与 Joiner 距离更近的一个。这个更近的定义是根据一个参数 `gmcast.segment` 来决定的, 优先选择和 Joiner 的 `segment` 相同的节点做 Donor, 而如果没有相同的 `segment`, 则直接选择一个其他的候选节点, 因为选择所用的算法是相同的, 同时每个节点中对所有节点的保存顺序相同, 所以选择结果都是一样的, 这样最终只会有一个节点是 Donor。

对于是不是每个节点都要继续做 Donor 的工作的问题, 在这里会做个判断, 如果被选为 Donor, 就继续向下执行, 否则就在这里停止了。

被选为 Donor 的节点, 其状态会被修改为 `DONOR/DESYNCD`, 这正如我们通过状态参数 `wsrep_local_state_comment` 所看到的一样。选择一个合适的 Donor 之后, 第一层处理就结束了。到第二层, 第一层处理后的消息交给第二层 `dispatch` 消息队列处理函数 `process_state_req`, 这个函数首先要做的工作是, 根据消息内容, 来确定 GTID 值及要做 SST 还是 IST。做 SST 还是 IST, 其行为会稍微有点不同, 这里首先说一下回调函数 `wsrep_sst_donate_cb`, 这个函数是 MySQL 上层定义的, MySQL 还会构造一个命令, 命令格式如下。

```
wsrep_sst_xtrabackup-v2 --role 'donor'
--address '127.0.0.1:4308/xtrabackup_sst' --auth 'sstuser:password'
--socket '/var/mysql/multi/3308/socket/mysql.sock'
--datadir '/var/mysql/multi/3308/data/'
--defaults-file '/var/mysql/multi/3308/etc/my.cnf'
```

```
--binlog '/var/mysql/multi/3308/binlog/mysql-bin'
--gtid 'fa394636-f604-11e4-83cc-ef3d70b1a3e3:48'
```

同样地，还会创建一个线程，名字为 `sst_donor_thread`，这个线程的作用就是去执行脚本 `wsrep_sst_xtrabackup-v2`，但这次是以 Donor 的身份去执行，在执行时，与 SST 或 IST 有关系，因为此时的 Joiner 端还在等待。

如果做 IST 的话，此时 Galera 在调用这个回调函数前，先判断要做 IST 的开始 GTID 值在 Donor 这边是不是存在。如果存在，则锁定这个在 Gcache 中的事务（以免 PURGE 掉了），然后调用回调函数，因为此时是不需要备份的，只需要通知 Joiner 不需要再等待了，做的是 IST，让它直接退出即可。脚本 `wsrep_sst_xtrabackup-v2` 的操作是，给 Joiner 传一个文件 `xtrabackup_ist`。而从脚本代码中可以看到，如果有这个文件，则它会直接将当前的 GTID 信息输出给上层 MySQL，然后就退出了；如果对应的 GTID 值在 Donor 中不存在，则说明 Donor 中 Gcache 缓存的内容已经不够给 Joiner 提供增量数据了，需要转换为 SST 的方式。

而如果是做 SST 的话，执行时首先会将 GTID 信息传给 Joiner，因为此时 Joiner 正在等待 GTID 的接收，超时设置为 100 秒，这个 GTID 正是执行这个回调函数时当前节点提交的最后一个事务 Xid 号。而在 Joiner 端收到的文件名为 `xtrabackup_galera_info`。接下来的操作，涉及 Joiner 与 Donor 之间的交互，在 Donor 传完 GTID 之后，Donor 主动等 10 秒钟，这是为了让 Joiner 准备好接收数据，也就是启动 Socat 或 NC 服务，这边在 10 秒钟之内应该可以启动完成，而 Donor 在 10 秒之后，就开始执行下面的命令。



```
innobackupex --defaults-file=/var/mysql/multi/3308/etc/my.cnf
--no-version-check $tmpopts $INNOEXTRA --galera-info
--stream=$sfmt $itmpdir 2>${DATA}/innobackup.backup.log |
pv -q -L $rlimit | socat -u stdio TCP:127.0.0.1:4308;
```

此时，Joiner 已经准备好接收了，这样两个节点之间的数据传输就会一直做，直到完成。传输完之后，Joiner 这边所需要做的就是 XtraBackup 的 APPLY 了，完成之后，因为 XtraBackup 已经做完，生成了新的 `xtrabackup_galera_info` 文件（必须要做完 APPLY 之后，才会生成这个文件），所以这个文件中存储的是当前传过来的数据中最新的 GTID 信息，脚本通过 `echo` 命令打印文件信息传给上层 MySQL 的 `sst_joiner_thread`（这一步与上面的 IST 方式对应起来了，IST 方式是直接将当前最新 Joiner 的 GTID 值告诉线程 `sst_joiner_thread`，用来表示当前 Joiner 节点最新的数据状态）。

从做 SST 开始到 SST 结束期间，集群中的线上节点有可能会有写入操作，这些数据在 Donor 端做 XtraBackup 时已经包含了（用 XtraBackup 的 `xbstream` 方式做备份），所以在处理接收队列时，如果队列中的事务 GTID 值比做完 SST 之后的最大 GTID 值小，那么这些事务会被忽略。不过，这个最大 GTID 值，Galera 是不知道的，上层 MySQL 还需要告诉 Galera，这

样 Galera 才可以判断哪些事务是需要 APPLY 的, 告诉 Galera 的方式是调用 Galera 的接口: `galera_sst_received`, 其参数就是 GTID 信息, 这也是线程 `sst_joiner_thread` 将 GTID 值传给 MySQL 层的意义所在。函数 `galera_sst_received` 在上面已经见过了, 作用是打一个标志, 告诉 Joiner, SST 已经完成, 可以处理后面的 IST 增量数据了。

SST 完成之后, Joiner 接收到的数据就是到这个 GTID 为止的。在 SST 收尾时, Joiner 再向集群发送一个类型为 `GCS_MSG_JOIN` 的消息, 这个消息每个节点都会收到。而在做完 SST 到发送 `GCS_MSG_JOIN` 消息之间的写入, 则会被正常 APPLY。做完这些之后, 才开始处理消息 `GCS_MSG_JOIN`。

GCS_MSG_JOIN 消息

因为每个节点都会收到这个消息, 而只有 Donor 及 Joiner 才会对它进行处理。在这里, 不管是谁收到这个消息, 处理方式都是一样的, 首先会被转换为一个新的 ACTION: `GCS_ACT_JOIN`, 然后将当前节点的状态从 `JOINING` 或 `DONOR/DESYNCED` 修改为 `JOINED`, 然后再向集群发送一个 `GCS_MSG_SYNC` 消息。

GCS_MSG_SYNC 消息

同样地, 这个消息如果是和正常数据写入一起处理的, 就会被转换为 ACTION: `GCS_ACT_SYNC`, 在 `dispatch` 中处理时, 其处理方式是调用 Galera 的回调函数 `wsrep_synced_cb`, 完成之后, Joiner 就正式加入了集群, Donor 又恢复为正常状态, 同时状态参数 `wsrep_local_state_comment` 也就都变为了 `Synced`。而关于回调函数 `wsrep_synced_cb`, 属于在 MySQL 中实现的接口, 就是同步之后做的一些收尾工作。

到此, 很费劲的 SST 就执行完成了。看起来可能比较复杂, 但知道大概过程之后, 可以让 DBA 知道它究竟在干什么, 打印出来的日志是什么意思, 已经走到了哪一步, 如果出错了, 是哪一步错了, 等等。

在即将完成本章写作的时候, 我自己不太放心, 折回去又读了一遍刚刚所写的东西, 突然眼前一黑, 真的感觉好乱啊! 我顿时手足无措。想想也是, 这个东西还是相当复杂的, 还是加一个图吧, 这样会直接一点, 无图无真相嘛。于是我花了半个小时又画了一幅图 (图 34.1), 请读者慢慢欣赏吧。

图 34.1 的一些说明如下。

- 为了更明确地表示消息发送接收的顺序, 给每一个动作都加上了序号。
- 中间的螺旋图表示的是虚拟集群, 也指消息发送的网络, 接收者其实都是实际的集群节点, 也是消息的发送者。

- 左边的节点用来表示与众不同的节点，比如新启动的节点、不同步的节点等，而右边的 0~2 号节点表示的是集群中的所有节点，实际的消息接收者，也是大部分消息的发送者（每一个都会发）。
- 在做 State Transfer 的过程中，Donor 和 Joiner 的状态都会变为不同步的，那么在做完之后，最后一步，这两个节点都会变为一致状态，也就是 23 步及 24 步所表示的内容。
- 图 34.1 中以 GCS 开头的状态或消息类型，只是表意而已，属于代码中的东西。

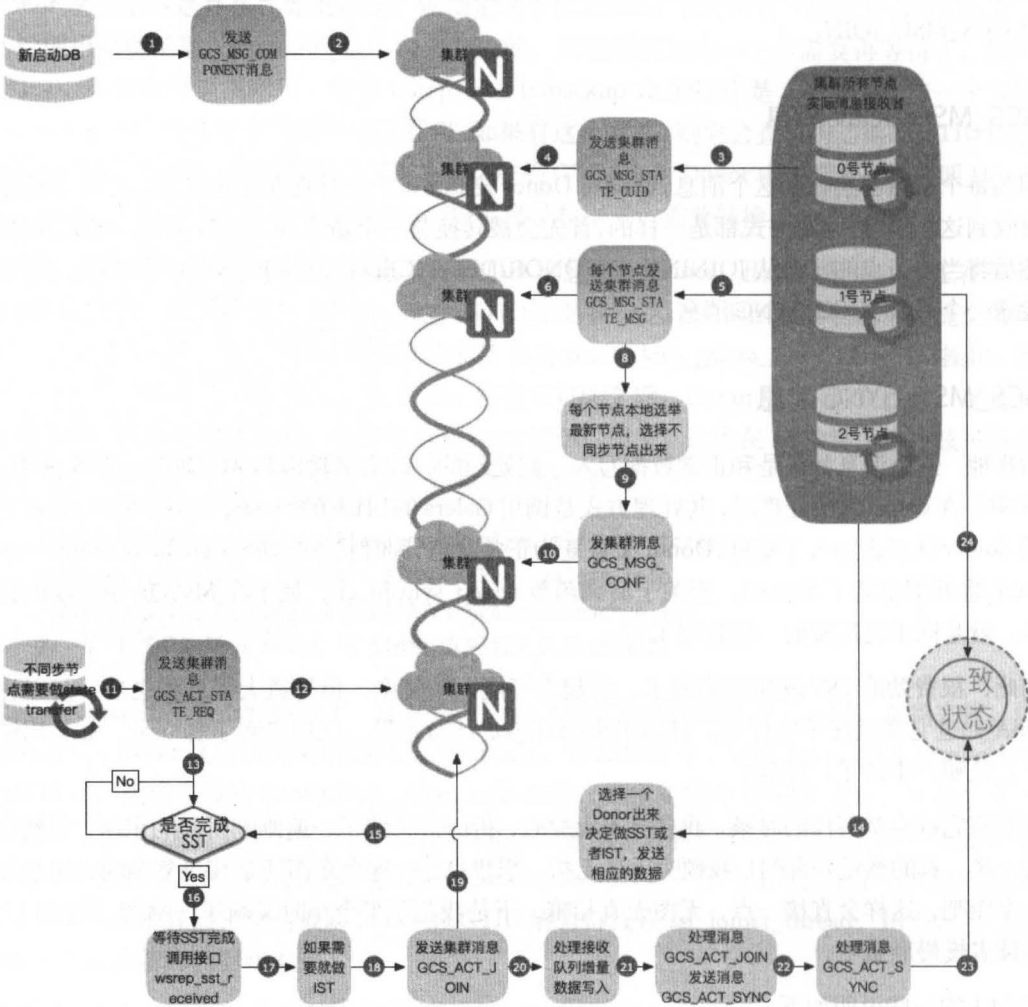


图 34.1

总之一句话，要“知其然，知其所以然”。

Galera 的代码也很复杂，本节内容中涉及很多函数名，主要是为了记录其中的过程，并为有兴趣的同学以后查看具体实现行个方便。

如何提供增量数据

前面已经讲述了 SST/IST 的整个过程，但还有一点没有讲述清楚，那就是如果要做 IST 的话，Donor 是如何给 Joiner 来提供数据的，而 Joiner 是如何处理这些增量数据的，这里作为补充再详细说明一下。

上面已经说到，如果 Joiner 选择做 SST，则 Donor 会直接做备份，然后将全量数据传给 Joiner，Joiner 接收到之后，再处理接收队列中的增量数据，这部分增量数据的处理方式和正常从节点（相对写节点而言）的 APPLY 数据方式没什么区别，只是会判断增量数据中是不是存在老数据，如果存在就不做 APPLY，否则就 APPLY。

而如果 Joiner 选择的是 IST，那么消息发送到 Donor 之后，Donor 通过判断，发现这个 IST 是可以做的，Donor 就会创建一个 asio 的发送线程，将 Donor 节点上面 Gcache 中的缓存事务从请求点开始，按顺序一个一个地向 Joiner 节点发送。现在已经知道，在 Joiner 决定做 IST 之后，它首先会选择等待 SST（如果实际做的是 IST，那只是一个通知操作而已，通过调用 `galera_sst_received` 来通知）完成，等待结束之后，再继续做 IST。不过这里有一点需要判断，前面做的是全量真实的 SST，还是只是一个通知信号的 SST。判断方法是，如果是全量 SST，则本地收到的 GTID 肯定大于等于 Joiner 请求 State Transfer 时 Donor 端的 GTID 值。因为这期间 Donor 还会继续写入，备份结束时的 GTID 肯定是大于或等于（Donor 没有写入）备份前的 GTID 值的，如果确实是大于等于，则不需要做 IST 了，说明是真实的全量 SST，而如果小于，则说明是一个通知信息的 SST（真实的 IST 方式），需要通过 IST 来接收新的增量数据。而此时 Joiner 会创建一个 Receiver 线程，和 Donor 那边的 asio 发送线程对应，一个接收一个发送，接收者把接收到的事务一个一个地 APPLY，接收完毕之后，Joiner 算是做完了 IST，然后继续处理后续消息，比如 `GCS_MSG_JOIN`、`GCS_MSG_SYNC` 等，等到做完后续的 Join 操作，最终会将状态从 Joining 变为 Synced。而 Donor 端也是一样，发送完成之后，也会处理这两个消息，逐步地将状态从 DONOR/DESYNCED 变为 Synced。

总结

- 不管是做 IST，还是做 SST，两边都需要执行 `wsrep_sst_xtrabackup` 等脚本，只是执行方式或过程不同而已。
- 因为两边都要执行脚本，而且脚本都是通过自己节点的 `wsrep_sst_method` 来指定的，我们现在有好几个版本，所以请在做之前，保证每个节点机器上面的指定脚本必须存在，不然会引来很多麻烦。

- 决定要不要做 SST 是在 Joiner 端决定的。靠的是文件 `grastate.dat` 中的信息 (UUID)，最终决定能不能做 IST，就是在 Donor 端决定的。
- Donor 的选择可以配置，不同机房可以通过在同机房的节点上，配置相同的参数 `gcast.segment` 来就近传输数据。
- 在 `wsrep.cnf` 配置文件中，`[sst]` 组中的 `time` 参数可以打开，这样在日志文件中就可以打印每一个步骤做了多少时间，有可能对 DBA 查询问题有好处。

35

Donor/Desynced 详解

这一章我们来讲一个很重要的细节，在命令 `show status like "wsrep%"` 的结果中，有一个状态参数为 `wsrep_local_state_comment`，有时候它的 `Comment` 为 `Donor/Desynced`，那这究竟是什么意思，是 `Donor` 么？如果是，那为什么还会有 `Desynced` 呢？

其实这个显示包括两种状态，只是这两种状态的处理是差不多的，所以一起显示了。

首先它表示的是 `Donor`。在做 SST 时，某一个节点在给要加入的节点 `Joiner` 提供数据时，这个节点就被称为 `Donor`，等新增节点数据同步完了，这个状态会变为 `Synced`。这种情况，这个状态参数就显示为 `Donor/Desynced` 状态。

可以让它显示为 `Donor/Desynced` 的另一种操作是通过设置全局参数 `wsrep_desync` 来实现。这个参数表示的是当前节点是不是要与整个集群时刻同步，这个参数有两个值，包括 `ON` 和 `OFF`。

执行如下命令。

```
 set global wsrep_desync=on
```

执行之后，就将当前节点的状态变为了 `Donor/Desynced`，那么它究竟在内部是怎么实现的呢？下面来看看基本的过程。

实现方式

首先在 MySQL 层设置这个参数时,调用了 Galera 的接口 `galera_desync`,这个接口所要做的就是向整个集群发送一个 `GCS_ACT_STATE_REQ` 类型的消息,当集群中的节点收到这个消息之后,首先会判断是不是 `desync` 的请求,如果是,并且发起者是 `Synced` 状态的,才能成为 `Donor/Desynced` 状态,否则不会成功,也就是说只有同步状态的节点才有能力成为 `Donor/Desynced`。如果判断满足,则每个节点都将发起者节点的状态修改为 `Donor/Desynced`,就是说,现在每个节点都知道发起者是 `Donor/Desynced` 了,包括自己。

然后,如果处理消息的节点是发起者自己,则接收线程继续处理当前消息,它直接将当前节点的同步状态修改为 `Donor/Desynced`。那么此时操作已经完成,发起者节点的同步状态已经修改,在其他节点中,将发起者节点的节点状态修改为 `Donor/Desynced` 即可。

意义何在

将状态设置为 `Donor/Desynced` 状态之后,表示节点数据不会保证与集群保持实时同步,如当前节点太慢,Worker 线程做不过来(不同步了),则可以通过参数来设置是不是向其他节点发送 FC 消息,这个参数为 `gcs.sync_donor`,参数默认值为 `no`,这也就是说,如果为 `Donor/Desynced` 状态,则不向其他节点发送 FC 消息。

实际上,参数 `gcs.sync_donor` 表示的是最大发送 FC 消息的状态。如果值为 `no`,那么最大发送 FC 消息的状态为 `GCS_CONN_JOINED`,对应的值为 1,而如果值为 `yes`,则最大发送 FC 状态为 `GCS_CONN_DONOR`,对应的值为 2。`GCS_CONN_DONOR` 比 `GCS_CONN_JOINED` 大,所以是不发送 FC 消息的,也就是说,这个参数控制的是,在节点状态为 `Donor/Desynced` 的情况下,这个节点要不要向集群发送 FC 消息。

这两个参数其实是很很有用的,如果数据库在运行过程中,发现某一个节点经常发送 FC,则说明这个节点做不过来,而又不想将它下线,导致过时之后还要做 SST,则此时完全可以先设置 `wsrep_desync` 为 `ON`,然后再设置 `gcs.sync_donor` 为 `no`,来解决这个问题。

在设置之后,如果问题已经解决,那么可以再将这个状态改回来,要执行的命令如下。

```
 set global wsrep_desync=off;
```

而如果之前没有将其设置为 `ON`,则 MySQL 是不允许设置为 `OFF` 的。

同样地,在执行上面命令时,MySQL 调用了 Galera 的接口 `galera_resync`。这个接口所做的操作是向整个集群发送一个 `GCS_MSG_JOIN` 类型的消息,在每个节点收到这个消息之后,都将节点状态(当前节点在本地对其他节点的状态的内存表示)修改为 `Joined`,那么每个节点都会打出日志“Member 0.0 (db10) resyncs itself to group”,而针对不同节点,具体操作是不同的。

如果是发起者在处理这个消息,则发起者会将当前节点的同步状态修改为 `Joined`,而此时当前节点还会判断,当前的接收队列的长度是不是已经小于了参数 `lower_limit`,如果是的话,再发送一个 `SYNC` 消息,每个节点收到之后,都将其内存中的对发起者节点的节点状态修改为 `Synced`,并且如果是发起者节点自己处理时,还需要多一步操作,就是将当前节点的同步状态修改为 `Synced`,那么此时整个集群又回归正常了。

如果是发起者之外的节点在处理这个消息,则有所不同,它们只需要将发起者节点在它们内存中的状态值修改为 `Joined` 状态即可,这样就完成了 `Join` 的处理。之后就是等待发起者节点发送 `Synced` 状态的请求了,处理方式是一样的,只是修改内存状态值而已。

到这里,这个参数的状态已经讲清楚了,但可能要注意下面的问题。

问答环节

问:在设置 `wsrep_desync` 为 `OFF` 之后,有可能产生的队列堆积会不会导致这个节点一下子向其他节点发送很多 `FC` 消息呢?

答:这是不会的。首先,上面讲了参数 `gcs.sync_donor`,说明 `Joined`、`Synced` 状态下会产生 `FC` 消息。在设置为 `OFF` 操作的讲解中,已经知道它是有这两个过程的,首先设置为 `Joined`,然后再设置为 `Synced`。仅从我们目前所了解的现象及原理来看的话,当设置为 `Joined` 之后,由于这个参数的原因,它是有可能产生 `FC` 的。

`Galera` 是用另外的方式做了特殊处理并解决了这个问题。当变为 `Joined` 状态之后,`Galera` 记录了当前执行队列的长度 `fc_offset`,然后再发送 `SYNC`,而这期间,发送 `FC` 消息的队列长度的判断是从 `fc_offset` 开始算起的,也就是说忽略掉当前队列的长度的,所以在状态修改过程中,`FC` 是不会产生的。

而能不能发送 `SYNC` 消息的判断方法是当前执行队列的总长度小于 `lower_limit` 的值时,才能发送。也就是说,这个值其实是不会超过 `fc_limit` 的,当 `SYNC` 成功接收并且处理之后,`Galera` 才会将 `fc_offset` 设置为 0,这时整个集群的状态就是完全一致的了,也就都跟上了。而如果消息队列的长度一直比 `lower_limit` 大,这个节点就会一直处理 `Joined` 状态,直到队列长度小于 `lower_limit` 为止。

在成功设置当前节点的状态为 `Synced` 之后再发生 `FC`,就另当别论了,可以以全新的一次 `FC` 来处理了,并且说明这个实例执行主库的事务又做不过来了。

36

Galera 的并发控制机制

现在已经知道，Galera Cluster 可以实现集群中数据的高度一致性，并且在每个节点上生成的 Binlog 顺序都是一样的，这与 Galera 内部实现的并发控制机制是分不开的。所有上层到下层的同步、复制、执行、提交都是通过并发控制机制来管理的。这样才能保证上层的逻辑性、下层数据的完整性等。关于 Binlog 的顺序问题，会在第 40 章中做详细说明。

在 29 章中，已经明确知道，并发控制主要是在接口 `galera_pre_commit` 中完成的，这个接口是 Galera 最重要的接口之一，这里面实现了最重要的复制、验证逻辑。目前，这个接口中包括的并发控制有以下几点。

数据复制

目前的 Galera 版本中，写集数据的发送是通过 Asio 的异步方式将数据广播出去。这个发送操作是串行的，是一个临界区，因为在每次发送之前，逻辑上还需要分片，并且每次发送完成之后，需要等待一个 GTID 的值，所以为了保证数据的一致性，这个发送操作需要串行。而这里有一点需要注意，如果当前集群处理 Flow Control 状态还没有解除，那么所有的写操作都会被阻塞在这里，等待 Flow Control 的解除，解除之后才能继续发送。

而下面这三种并发控制，与上面数据复制的并发控制不太一样。这三种实现方式很类似，每种并发控制也都是一个临界区，只是进入临界区的条件不同，这个条件先被称为 `condition`。也就是说，满足这个条件，就可以进入临界区，否则必须要等待。进入称为 `enter`，出来

称为 leave。相关参数可以参考 `wsrep_apply_oooe`、`wsrep_apply_ool`、`wsrep_commit_oooe`、`wsrep_commit_ool`，关于这些参数的意义，在第 30 章有专门讲述。

写集验证

LocalOrder：这个层次的 condition 为当前事务的 GTID，要比上一个已经验证通过的 GTID 值大 1。

这个条件的意思是，上一个已经成功处理的 GTID 要比当前事务的 GTID 小 1。那很明显，就是要求所有进入处理区的 GTID 必须是顺序的，因为 GTID 是顺序产生的，所以在顺序的基础上，同一时间必须只有一个事务可以进行处理，说白了，就是串行。

受这种层次并发控制管理的操作主要有验证操作，因此说验证是串行的。

写集 APPLY

ApplyOrder：这个层次的 condition 为当前事务依赖的 GTID 小于最新处理的 GTID 值，或者是本地事务。也就是说，当前事务所依赖的 GTID 已经被处理过了，则当前事务就可以被处理了，如果不满足这个条件，就需要等待。那么如果当前所有的事务都是没有关系的，也就是说验证之后，没有发现相互有依赖关系的事务，那就是并行执行的了。如果都是相互依赖的，就是串行操作了。

受这种层次并发控制管理的操作就是执行，并且是从库执行，也就是所说的 APPLY，从名字即可看出来。而这里涉及的并发数，就是 `wsrep_slave_threads` 所起的作用。

这个临界区是在接口 `galera_recv` 这个场景中被使用的。因为在验证阶段，当前事务会找是否存在一个它所依赖的事务，这是判断是否可以并行执行的条件，所以在从节点 APPLY 时，如果存在这么一个事务，就必须等待这个事务执行完成，它自己才能继续执行，也就是它才能进入这个临界区，否则就会等待，而如果不存在这样的事务，就可以直接 APPLY，也就是说可以并行执行。

在接口 `galera_pre_commit` 中，这个临界区也是有的，只不过此时当前事务已经执行完成，所以在验证时，肯定不会存在本地事务之间的依赖关系，因为都已经执行完了。到了提交阶段，各自所需要的资源都已经获取到了，它们之间就不会存在冲突关系了，所以只可能是本地事务与远程事务的依赖关系。

因此，APPLY 的串行只影响从节点（相对）执行从其他节点复制过来的事务，在主库执行本地事务时，不受影响。这就是 Galera Cluster 中的并行复制概念。

事务 Commit

CommitOrder: 这个层次的 condition 与一个参数有关系, 即 `repl.commit_order`。之前已经见到过这个参数了。

`repl.commit_order`: 它的默认值为 3, 也就是 `NO_OOOC` (NO out of order commit), 意思就是串行操作, 此时的 condition 是当前 GTID 必须要比最新处理的 GTID 大 1, 这和上面的 LocalOrder 是相同的。也就是说, 如果设置为 3 的话, 所有的事务提交都会是串行的, 在 Binlog 文件中, 看到的 Xid 都是依次增长, 步长为 1。这样的好处是, 所有节点的 Binlog 内容都是一样的, 如果集群下面再挂 Slave 的话, 它的主库 down 掉了, 就可以根据 Xid 来找到位置, 继续复制, 非常方便, 这点在第 40 章中介绍。

这个参数还有另外三个值, 如下。

- `LOCAL_OOOC` (LOCAL Out Of Order Commit): 对应的值为 2, 表示本地可以乱序提交。此时的 condition 是, 如果是本地产生的事务, 则条件成立, 而如果是远程产生的事务, 则还是串行执行的。
- `OOOC` (Out Of Order Commit): 对应的值为 1, 表示根本不受这个并发控制的限制了, 所有提交都是乱序即可。这样的并行提交与单点 MySQL 是相同的机制。但如果设置为这个值的话, 带来的后果是, 如果给集群加从库, 某一个节点为主库, 那么当这个主库挂掉之后, 这个从库也就挂掉了, 因为这个复制的点根本就没办法找到了。而如果是全顺序的话, 则可以随便在三个节点之间进行切换。
- `BYPASS`: 对应的值为 0, 表示所有的提交并发控制的检查都失效了, 也就是说这个条件永远为真。这和 `OOOC` 是一样的, 只是性能更好一些, 它主要是用来测试的。

讲清楚这个参数, 也就讲清楚了 this 层次的并发控制机制, 默认是 3, 建议也是 3, 就是串行提交, 这样就保证了不管在主库还是从库, 所有的节点产生的 Binlog 都是完全相同的。

以上就是 Galera 中关于事务操作的并发控制, 理解了这些内容, 也就对整个 Galera 的执行过程中关于数据一致性、完整性的控制, 有比较深入的理解了。

37

Galera 的流量控制

流量控制的定义

本节先来介绍一下流量控制 (Flow Control) 的定义。

在 Galera Cluster 中, 有一个参数叫 `gcs.fc_limit`, 它的全名其实是叫 `flow control limit`, 顾名思义, 是流量控制大小限制的意思, 它的作用是什么呢?

因为 Galera Cluster 是强同步集群, 所以在数据同步上面, 有各种机制保证数据延迟不要太大, 方法包括发送写集时等待本节点接收到 GTID 值, 确保本地消息已经收到, 并且在从节点, 通过多线程并行执行, 可以尽可能快地将数据复制完成。那么实际上, 效果也确实很好。通过深度测试发现, 在硬件资源、性能都一样的情况下, 数据延迟一般是在 10ms 之内, 大部分都是在 5ms 之内, 可以说这个值是非常可观的, 主从同步基本上都达不到这个值。而反过来讲, 完全的同步复制没有延迟是不可能的, 或者即使做到, 在性能方面的损失也是非常大的, 所以这个范围是可以接受的, 并且相当可观。

一个集群如果保持良好运行的状态, 并且硬件资源稳定, 性能平衡, 都可以将延迟保持在一个可接受的范围内。如果一套集群中, 由于某个节点或某几个节点的硬件资源比较差, 或者节点压力比较大等原因, 导致复制效率降低, 从而使得从节点 APPLY 时非常慢, 也就是说, 主库在一秒钟之内做的操作, 从库有可能会用 2 秒才能完成, 那么这种情况下, 就会导致从节点执行任务的堆积, 以及接收队列的堆积。

假设从节点真的堆积了,那么 Galera 会让它一直堆积下去么?这样延迟会越来越严重, Galera Cluster 就相当于变成了一个主从架构的集群,已经失去了强一致状态的属性。很明显, Galera 是不会让这种事情发生的,那么它是如何实现的呢?继续往下看。

此时,就要说到开头提到的参数 `gcs.fc_limit` 了,这个参数是在 MySQL 参数 `wsrep_provider_options` 中来配置的,而参数 `wsrep_provider_options` 是 Galera Cluster 的一个参数集合,其中流量控制 (Flow Control) 相关的其他参数还有 `gcs.fc_factor`。这两个参数的意义是,当从节点堆积的事务数量超过 `gcs.fc_limit` 的值时,从节点就发起一个 Flow Control,而当从节点堆积的事务数小于 `gcs.fc_limit * gcs.fc_factor` 时,发起 Flow Control 的从节点会再发起一个解除的消息,让整个集群再恢复。

说句题外话,由于 Galera 是 MySQL 的一个插件,虽然有些参数可以在以 `wsrep` 为首的 MySQL 参数列表中进行设置,但是还有很多参数是通过 `wsrep_provider_options` 这个参数集合来设置的,需要特别关注。在很多地方都会讲到这里面要设置的一些细节,请读者多加留意。

流量控制的实现原理及影响

Galera Cluster 流量控制的具体实现方式,会分两方面讲。

在从节点上,写集接收线程在收到每一个事务之后,都会检查当前接收队列的大小,如果当前队列长度大于所设置的 `gcs.fc_limit` 值,则当前节点会构造一个 `GCS_MSG_FLOW` 消息。现在已经知道, Galera 消息的处理原则是,无论是谁给集群发消息,每一个节点都会收到这个消息,只不过是发送者和接收者接收到信息之后的处理方式不同而已。那么此时,当前从节点发送 FC 之后,就继续处理其他消息,并且将接收到的消息都交给 `disptach` 线程去处理,这个节点只是在发现堆积之后,在处理消息之余发送了一个 FC 消息,这样每个节点收到之后,就会分别处理各自的这个消息。

在其他节点上,接收到消息 `GCS_MSG_FLOW` 之后,先记录下当前时间点,表示从什么时候开始暂停的,然后将当前节点的状态设置为暂停,那么此时,这个节点针对这个消息要做的事情就做完了。而这个动作对这个节点所带来的影响是什么,在第 29 章已经讲过。其实流量控制实质上是对发送消息的控制,当设置为暂停之后,如果这个节点再有新的写集复制请求,也就是有新的事务调用了 `galera_pre_commit` 接口,就会在复制 (广播) 这个位置被阻塞掉。同样,所有新的写入请求都会被卡在这里,直到原来发送 FC 消息的节点再发送一个 FC 的 `Continue` 消息过来,被阻塞的复制操作就可以蜂拥而上地做复制了。

上面说到在开始暂停时记录下来一个时间点,从这个时间点到 FC 恢复之间的时间,会被累计叠加到状态参数 `wsrep_flow_control_paused_ns` 中,表示暂停时间又增加了。这个参数也是针对 FC 的重要参数之一,也是需要重点监控的参数,如果真出现了,就需要重点关注这个问题,有可能出现的原因包括以下五方面。

- 发送 FC 消息的节点，硬件有可能出现了问题，比如 IO 写不进去或很慢，以及 CPU 异常高等。
- 发送 FC 消息的节点，本身数据库压力太高，比如当前节点承载太多的读，导致机器 Load 高，IO 压力大等。
- 发送 FC 消息的节点，如果硬件压力都没有太大问题，仍然做得比较慢，那么一般原因就是主库并发高，但从节点的并发跟不上主库。那么，此时可能需要观察这两个节点的并发度大小，可以参考状态参数 `wsrep_cert_deps_distance` 的值来调整从节点的 `wsrep_slave_threads`，这样应该是可以解决或缓解的。这个问题可以这样去理解，假设集群中每个节点的硬件资源都是相当的，那么主库可以执行完，从库为什么做不过来呢？一般思路就是像处理主从复制的延迟问题一样去处理。
- 检查存不存在没有主键的表，因为 Galera 的复制是行模式的，所以当存在这样的表时，由于主节点是通过语句来修改的，比如一条更新语句更新了全表，而从节点收到之后，就会针对每一行的 Binlog 做一次全表扫描，这样就会导致这个事务在从节点执行，比在主节点执行慢十倍，甚至百倍，从而导致从节点堆积进而产生 FC。
- 从节点执行了上锁操作，导致新的写集过来之后都不能执行，然后堆积进而出现了 FC。

以上可以看出，产生 FC 的原因很多，这对 DBA 来说不是什么好事，因为这些原因都需要 DBA 去处理、解决。在第 15 章中，讲述了很多处理延迟的方法，可以参考。

从节点发送 FC 之后，其他节点都收到了，此时对整个集群产生的影响是不言而喻的。在状态变量上，除了上面提到的暂停参数之外，还包括 `wsrep_flow_control_paused`、`wsrep_flow_control_sent`、`wsrep_flow_control_recv`，这些也都是需要重点监控的，如果发现增加了，则说明都是有问题的，需要尽快解决。

上面已经讲到了解决方法，那么在解决之前，首先需要找到究竟是哪个节点发送的 FC 消息，这一般需要下面两个状态参数来帮忙。首先来看 `wsrep_flow_control_recv` 参数，表示的是从上次 flush status 开始，一共收到了多少次 FC 消息。上面已经讲过，不管谁发送了 FC 消息，每个节点都会收到这个消息，也就是说，每个节点的 `wsrep_flow_control_recv` 都会增加，因此通过这个参数判断不出来是哪个节点发送的。所以，就要通过参数 `wsrep_flow_control_sent` 来判断了。是哪个节点发送的，其对应的这个参数就会增加，而其他节点不会变化，所以根据这个参数的监控就可以很快找到比较慢的节点，然后进一步解决这个问题。

说到这里，其实此时整个集群还是处于暂停状态，解决问题要紧。

这个问题还需要问发起节点，正所谓解铃还需系铃人。

我们已经知道，接口 `galera_recv` 是负责处理接收队列中的事务的，其线程的个数等同于 `wsrep_slave_threads` 的值，接收队列的实时长度，每个线程都是最清楚的。所以，每次在从队列中取到一个事务之后，都会检查当前是否正处于流量控制的暂停状态，如果是，则

再判断 `gcs.fc_limit * gcs.fc_factor` 是不是大于当前接收队列的长度, 如果是, 则发送消息 `GCS_MSG_FLOW`, 其消息内容是 `GCS_FC_CONT`, `CONT` 表示的是 `CONTINUE`, 这样集群中的每个节点都会收到这个消息, 收到之后, 将各自节点的状态修改为非暂停状态, 然后统计一下从暂停到收到 `CONTINUE` 之间所用的时间纳秒数目, 累加到状态参数 `wsrep_flow_control_paused_ns` 中。然后, 将所有正在被阻塞的连接唤醒, 此时整个集群就又涛声依旧了。

两个问题

问: 所说的队列长度的单位是什么?

答: `fc_limit` 的单位是事务的个数, 与事务大小没有关系, 每增加一个事务, 长度就加 1。

问: 灵敏度可以控制吗?

答: `gcs.fc_factor` 的作用是一个恢复比例, 是相对 `gcs.fc_limit` 而言的, 但是从运维的角度而言, 为了最大限度地提供服务, 一般建议 `gcs.fc_factor` 设置为 1.0 即可, 也就是说队列长度只要小于 `gcs.fc_limit`, 集群的写入即可恢复, 而不是等消费到一定程度才恢复, 这样相对减少了不可服务时间。同时这样设置的时候, 流量控制相对会更加灵敏, 可能会频繁出现流量控制, 而恢复得也非常快。当然, 如果出现一个大事务导致一时的流量控制, 那么只要这个事务执行完了, 马上就可以恢复, 从而可以在一定程度上减少这种问题的影响, 所以设置为 1.0 应该比较好的。

Galera Cluster 影响单节点执行效率的因素

经过大量的测试发现，Galera Cluster 单个节点执行 SQL 时，执行的 QPS 会受到影响，让人感觉比较慢，这与非 Galera Cluster 架构的单实例 MySQL 的性能会有一定的差距。那么，这之间的缘由究竟是什么呢？我们单独用这一章的内容把这个问题说清楚。

单点验证

现在已经知道，即使是单个节点写入，在这个节点上面的所有写集（Write-Set）之间也需要做验证。不过笔者原来误认为，这部分的本地认证（所有本地会话发出的写请求）其实是不需要的，本节点需要做的验证只是本地写集与远程修改发送到本地写集之间的验证。因为当本地会话的所有写集走到验证这一步的时候，就说明已经没有冲突了，需要修改的都已经修改完成，只差提交这一步了，所以不需要验证。但是这样的想法是错误的。

经过研究发现，即便是单个节点写入，在这个节点上面的所有写集之间还是需要验证的。所以，第一个引起慢的原因就是本地写集验证。

没有冲突为什么还会验证呢？因为 Galera Cluster 是多点写入的，不能只考虑一个节点，在某一个节点写入的同时，其他节点也有可能在写入，验证区就有可能存在其他节点写入的事务，虽然本地事务之间是不冲突的，但验证过程还是需要做的，这个验证主要是为了防止本地写入与其他节点写入的冲突。

并发控制

并发控制方面，首当其冲要考虑的就是写集的发送。当 Galera 拿到事务的写集之后，通过接口 `galera_pre_commit` 来将其广播到整个集群中，这个发送操作进入的是一个全局的临界区，也就是说发送写集是串行的。经过测试发现，这个临界区对 Galera 的性能影响最大，并且在网络不稳定的时候，影响更大一些，并且在 QPS 方面的体现很不稳定，都是与这个有关系的。

通过了解之前讲过的 Galera 验证方法的相关内容，现在已经知道，Galera 为了处理多点写入的冲突验证检查问题，采取了很多并发控制策略，耳熟能详的有写集的验证过程，即每一个提交的事务都会在当前节点的验证区中验证，验证过程的临界区是串行的，这对 Galera 的性能影响比较大。

另一个并发控制的串行区是提交操作，因为参数 `repl.commit_order` 默认并且建议设置为 3，这样带来的唯一弊端就是提交操作变成了串行，但保证了 Binlog 的顺序，给运维带来了很大的便利性，但串行又会导致 Galera 的性能有所下降。

等待 GTID

对于 Galera 的实现，目前已经基本清楚了。在某一个节点上，每提交一个事务，在将写集发送出去之后，都会等待一个通知，这个通知就是告诉这个事务，GTID 已经产生了，这个 GTID 类似这个事务的身份证，是一个终生不变的标签。这个等待过程，也是影响 Galera 性能的一个重要指标。

总结

以上三部分就是现在所认为的在 Galera Cluster 架构下，即使是单节点执行效率也不及单实例 MySQL 的原因。

慢与快，是相对而言的。俗话说：“强中自有强中手。”快也是一样的道理，没有最快，只有更快。对于 Galera 而言，虽然本章所讲的是什么影响了 Galera，导致它性能被限制了，但是相对单点 MySQL 来说，确实是慢了。而对于不是极端写入压力非常大的业务，Galera Cluster 已经很够用了，因为它的强同步机制及多点写入机制，给运维及业务带来的好处无以言表，所以损失一些性能是可以接受的，这是够用的，特别是对于交易类型的业务而言，是非常好的选择。

关于性能上的具体数据，可以参考一些网站上的结果。当然，还是要自己测试才最有说服力，可以准确地判断 Galera Cluster 是不是适合你。

还是想说，够用就好！

39 grastate.dat 文件揭秘

引子

众所周知，在基于 Galera Cluster 架构的 MySQL 数据库正常关闭时，都会生成一个完整的 grastate.dat 文件，文件内容一般如下。

```
# GALERA saved state
version: 2.1
uuid: 6e69d929-8427-11e5-8b0f-3a5332eb882b
seqno: 123456789
cert_index:
```

文件中的 uuid 表示当前节点所属集群的 wsrep_cluster_state_uuid 值；而 seqno 是当前节点在整个集群中的状态，也就是 wsrep_last_committed 的值，也是节点下次启动做 IST 的初始点，而如果是非正常关闭，或者处于运行状态时，这个值便为-1。

在之前对 Galera 的研究中认为，Galera 启动时做 IST 的初始点是从 grastate.dat 中取出来的。当然，这到现在为止还是成立的。只是最近发现了一种新情况，那就是当数据库正常关闭之后，把 seqno 改为-1，或者在启动的时候，由于某种原因失败了，导致 seqno 被冲掉变为了-1，这种情况下，可以通过以下方式来自正常启动。

```
sh /etc/init.d/mysql.server -P 3306 start
```

这样执行之后，竟然发现可以很快通过 IST 方式加入到整个集群中。

分析研究

这种情况是之前没有考虑到的,而现在发现了,那就让我们来弄明白在 Galera 中,是如何处理这种情况的。

首先,我们知道,启动脚本 `mysql.server` 的实现,其实是执行了 MySQL 自带的启动脚本 `mysqld_safe`。从启动脚本文件 `mysqld_safe` 中发现,有专门的地方处理了 Galera 的启动,如果文件 `grastate.dat` 存在,就找到文件中 `seqno` 对应的值,如果这个值不是 -1,那么就说明这个数据库实例是正常关闭的,直接启动即可。而如果这个值为 -1,那就要通过下面的步骤来启动。

1. 在数据目录中创建一个临时文件,名字为 `wsrep_recovery.XXXXXXX`。
2. 指定一个 `pid` 文件,用来临时启动 `mysqld`。
3. 在日志中打印如下信息。

```
151106 14:48:10 mysqld_safe WSREP: Running position recovery with
--log_error='/home/mysql/multi/3308/data/wsrep_recovery.gTUZcN'
--pid-file='/home/mysql/multi/3308/data/l-host-name-recover.pid'
```

可以看到,生成的临时文件为 `wsrep_recovery.gTUZcN`,而 `pid` 文件为 `l-host-name-recover.pid`。

4. 然后执行如下命令。

```
mysqld --defaults-file=/home/mysql/multi/3308/etc/my.cnf
--log_error='/home/mysql/multi/3308/data/wsrep_recovery.gTUZcN'
--pid-file='/home/mysql/multi/3308/data/l-host-name-recover.pid'
--wsrep_recover
```

启动时指定了参数 `wsrep_recover`,这是专门用来获取 GTID 信息的,获取完成后, `mysqld` 就退出了。当然,如果之前这个数据库实例是非正常关闭的,那么执行这个命令时,也会做数据库的恢复操作。


5. 接着在生成的临时文件 `wsrep_recovery.gTUZcN` 中搜索信息 `WSREP: Recovered position:`,在这个字符串之后紧跟着的,就是 `UUID` 加上 `seqno` 信息。
6. 最后将上面得到的 GTID 信息给 `mysqld` 的参数 `wsrep_start_position_opt`(就是参数 `wsrep_start_position`),再启动数据库。此时数据库就会以 `IST` 的方式启动。

总结

从上面的步骤中,可以总结出如下五个特点。

- `grastate.dat` 文件必须要存在,如果不存在,则肯定会做 SST。

- grastate.dat 文件在上面的步骤中,看上去,数据库实例正常关闭后,如果手动修改文件,即使 uuid 为 00000000-0000-0000-0000-000000000000, seqno 为 -1,似乎也不会出问题,也会正常启动。但事实却不是这样的,通过看 galera 的源码发现,在启动时通过参数 `wsrep_start_position_opt` 获取到 GTID 信息之后,galera 判断当前 uuid 是合法的,同时 seqno 为 -1,且 grastate.dat 存在,这些条件都满足的时候,才会使用 `wsrep_start_position_opt` 参数中的 seqno 值。也就是说,如果 grastate.dat 文件存在,但 UUID 不合法,则 galera 就会将 `wsrep_start_position_opt` 中的 seqno 部分忽略掉,转而做 SST 了。所以,grastate.dat 文件中的 UUID 必须与集群的 UUID 一致。
- `mysqld_safe` 临时启动了一个 `mysqld`,还添加了参数 `--wsrep_recover` 和参数 `--log_error`,指定了错误日志文件。通过阅读 `mysqld` 的代码发现,如果指定了参数 `wsrep_recover`,其实并不能真正地启动数据库,而是调用了函数 `wsrep_recover` 之后就退出了,并且将 GTID 信息打印到指定的错误日志文件中。打印格式如下。

 2015-11-06 10:52:37 9152 [Note] WSREP: Recovered position: 6e69d929-8427-11e5-8b0f-3a5332eb882b:6

从而有了上面的第 5 步,目的就是要得到后面的 GTID 信息。

- 关于 `wsrep_recover` 函数的作用,主要是从 InnoDB 文件中读取了在每一个事务提交时,写入的一个 `wsrep` 检查点,检查点的信息就是 GTID 信息。这样,GTID 就可以保证是最新成功提交的一个事务,在启动时使用这个值就是一个正确的点。
- 如果以后正常关闭的数据库启动失败,导致 seqno 丢失,不要紧张,也许还是有救的。此时如果心里有底的话,直接使用 `/etc/init.d/mysql.server -P 3308 start` 方式启动;如果心里没底的话,可以尝试一下上面的第 4 步,看看数据库中有没有写入正常的 GTID 值,如果有,可以直接将 GTID 拿出来,写入到 grastate.dat 中,这样 DBA 可能会觉得心里更有底吧。当然,这也是一种找 GTID 值的方法,这才是关键!

40

Galera Cluster 从库的转移

在深入理解 Galera Cluster 之前，一直有一个疑问，既然 Galera 是多线程复制的，那么不同节点的 Binlog 顺序应该是不同的，至少是不尽相同的。如果真的是这样，那么当某一个 Galera Cluster 节点挂掉之后，在这个节点上挂的从库也就不能用了。因为节点之间的 Binlog 记录顺序不一致，导致了无法正确地其他节点找到对应的 Binlog 位置或下一个要执行的语句，即便是找到了，也无法确定该位置前后的语句是否已经执行，这个问题困扰了我很久。当然，也许有人没有考虑过这个问题，觉得这压根不是问题，想当然地就把从库从一个节点转移到另一个节点了。其实，本质并没有那么简单。

在深入研究 Galera 之后才了解到，Galera 的作者在设计之初的确考虑过这个问题，并且也很周到地给出了各种方案，可以通过配置参数来满足我们的需要。确切地说，可以通过配置 `repl.commit_order=3` 保证所有节点的提交都是串行的，并且是以 Galera 层的 GTID 为顺序的。Binlog 的写入是在提交阶段进行的，因为提交是串行的，所以 Binlog 的写入也是串行的。这样一来，每一个节点上 Binlog 的顺序都是一样的。由此可以得知，从库是可以在不同节点之间游走的。

既然可行了，那么这里讲一下 DBA 的操作步骤，需要分两种情况，包括开启了 MySQL GTID (MySQL 5.6 及以上版本，GTID 为 Server 级) 功能与没有开启这个功能两种情况。

先来做一个约定。因为 MySQL 和 Galera 的 GTID 值格式是一样的，都包括两个部分，分别是 `gtid_uuid` 和 `gtid_seqno`，后面就以 `gtid_uuid` 来表示 GTID 的前半部分，用 `gtid_seqno` 来表示 GTID 的后半部分。

没有开启 Server 级 GTID 的情况

在没有开启 Server 级 GTID 的情况下，Galera Cluster 从库转移的步骤如下。

1. 主库挂了，从库的 Binlog 位置已经不会再发生变化，首先找到当前主库的 Binlog 位置信息：Relay_Master_Log_File 及 Exec_Master_Log_Pos，或者找到当前从库执行到的位置信息：Relay_Log_File 及 Relay_Log_Pos。这两个位置对应的 Binlog 内容是完全相同的，所以都可以用。
2. 通过 mysqlbinlog 工具，并根据上面已经找到的 Binlog 文件及位置信息，找到对应的 Binlog 内容。

如图 40.1 所示的内容，是根据位置找到对应的 Binlog。需要注意的是，图中显示的 Binlog 位置是 59969，但 end_log_pos 对应的值为 59837，大小关系不正常。造成这种现象的原因是，从 Relaylog 文件中取出来的 Binlog 内容，与主库产生的 Binlog 内容是错位的，因为在生成 Relaylog 时是在从主库 Dump 过来的 Binlog 内容前面加上一些头信息，一般相差一百多个字节，这一点大家可以自己研究。这里更重要的信息是 Xid，正是当前从库已经执行到的逻辑上的全局 ID，也就是 Galera Cluster 中的 GTID（Galera 级）的后半部分——gtid_seqno。

```
# at 59969
#150204 11:03:13 server id 29263 end_log_pos 59837 CRC32 0xf959c951 Xid = 1908230958
COMMIT/*!*/;
```

图 40.1

3. 选择合适的新从库，当然尽量选择只读节点即可。实在没办法，就选择写节点，影响并不大。
4. 找到这个位置之后，说明此时从库已经将这个位置的事务执行完成，这里的目的是要找到这个 GTID（Galera 级）。找到这个 GTID 之后，新的任务就是要找到在其他节点上面，这个 GTID（Galera 级）所在的 Binlog 位置，可能有点困难，但必须要找到，方法就是大致估计 GTID（Galera 级）所在的 Binlog 文件，然后通过 mysqlbinlog 解析逐个搜索，直到找到为止。
5. 找到对应的 Xid 所在文件之后，要选择新主库的 Binlog 位置，上面的显示是 1908230958。那么接下来就要找到下一个 GTID（Galera 级），因为这是已经执行过的 GTID（Galera 级），所以下一个明显就是 1908230959。但我们要的不是下一个 GTID（Galera 级），而是下一个 GTID（Galera 级）对应的事务在 Binlog 中的开始位置。其实很容易，当找到 Xid 时，也就找到了事务提交的事件，可以发现每个 Xid 后面紧跟的就是 COMMIT 语句。那么下一个事务就是紧跟着的 #at xxxx，如图 40.2 所示。


```
# at 59969
#150204 11:03:13 server id 29263 end_log_pos 59837 CRC32 0xf959c951 Xid = 1908230958
COMMIT/*!*/;
```

图 40.2

现在可以知道，要设置的新的 Binlog 位置就是当前文件的 60000 这个位置。

6. 执行 `change master xxx;start slave;` 命令即可。

开启了 GTID (server 级) 的情况

在 PXC 版本中，如果目前开启了 GTID (server 级)，是没什么用的。因为 GTID 的好处是在一个集群中，将每一个事务都标记一个唯一的 ID 值，这样在复制过程中，就可以通过这些 ID 集合，判断是否执行过了，从而实现尽可能高的容错性及尽可能简单的操作。但这主要是在主从复制中发挥作用，在 Galera Cluster 中，目前还没有太大的作用。

实际上，在 MySQL 运维过程中，如果不是为了完成从库在不同主节点之间来回切换，开启 server 级的 GTID 是没什么好处的，因为主库之间的同步，不是靠传统的主从复制来做的。但可能有些同学会注意到，同一个事务对应的 Binlog 内容的 `SESSION.GTID_NEXT` 的值在 PXC 集群的多个节点之间是一样的，所以进而以为从一个节点的复制，指到另一个节点时，不需要做任何修改，将 `auto_position` 设置为 1，就可以直接无缝切换到另一个节点。

而如果想要证明这一点，有时候就会去核对 Binlog 内容。而在核对之后，发现是这样的情况，事务内容是一样的，`SESSION.GTID_NEXT` 也是一样的，则有可能是对的，但即使发现是这样，就能证明这个肯定是对的么？

因为正好对得上，所以有没有可能是碰巧呢？比如在执行的时候，语句都完全一样，执行了 100 个，假设这 100 个事务对应的 `SESSION.GTID_NEXT` 值都是从 1 到 100，那么在 Binlog 内容中，这 100 个事务的内容是完全一样的，那此时如何来核对是否对应呢？有没有可能是集群中的几个节点存在 `SESSION.GTID_NEXT` 值的错位？比如说，第一个节点中第 50 个事务对应的 `SESSION.GTID_NEXT` 值是 50，而第二个节点中第 60 个事务对应的是 50，那么从第一个节点的复制转到第二个节点时，就产生了丢失 10 个事务的情况，从而导致数据不一致。

可能有人会问，怎么会产生这样的问题？

你可以在某个节点上执行一条语句：`reset master`，然后再继续执行一下，看一下生成的 Binlog 里面的 `SESSION.GTID_NEXT` 变成了多少，此时你可能就明白了。这种 `SESSION.GTID_NEXT` 与事务在集群之间不对应的情况是有可能发生的，人工核对基本上很难核对准确。所以上面就说，“有可能你是对的”，但这里是存在可能性的。

既然 SESSION.GTID_NEXT 在集群的多个节点之间，不能完全保证一致性，那么这个根据 GTID (server 级) 在不同主节点之间做随意切换的路是行不通了，因为错位的问题有可能出现，那也就存在丢数据的可能性，当然也有重复执行的可能性。

不过，可喜的是，经过对这两个关于 GTID 在 Galera Cluster 中产生机制的研究发现，实际上有希望让 Server 级的 GTID 在产生时就直接将 Galera 所生成的 gtid_seqno 来作为自己的 gtid_seqno。这样在任何时候，Xid 的值就和 SESSION.GTID_NEXT 的 gtid_seqno 是一样的了。那么，此时从库的切换就非常方便了，因为集群中的所有节点产生的 Binlog 内容都是完全一样的，包括 SESSION.GTID_NEXT 的值（目前可能还不是，可以通过 reset master 重现出来）。这样的话，从库在不同节点之间的切换就可以完全随自己的心情了，想切换就切换，想换就换。

不过，这个解决方案已经给 Percona 提出来了，回复是在下一个大版本中有可能加入。目前也正在和 MariaDB 沟通，本人通过调研并修改代码发现是完全可行的，静候佳音吧。

如果目前想要实现这个功能，自行修改一下代码即可。在文件 sql/rpl_gtid_cache.cc 中，找到函数 generate_automatic_gno，对比修改之前的代码，如下。

```
#ifdef WITH_WSREP
if (WSREP(thd) && thd->wsrep_trx_meta.gtid.seqno != -1)
{
    /* 在改之前，GTID前半部分使用的就是Galera层的GTID字符串 */
    automatic_gtid.sidno= wsrep_sidno;
}
else
{
    #endif /* WITH_WSREP */
    automatic_gtid.sidno= gtid_state->get_server_sidno();
    #ifdef WITH_WSREP
    }
    #endif /* WITH_WSREP */
    gtid_state->lock_sidno(automatic_gtid.sidno);
    /* 但是后半部分使用的就是server层的GTID值了，那么在Binlog中，
       SESSION.GTID_NEXT值就变成了galera_uuid:server_seqno这样的内容 */
    automatic_gtid.gno=gtid_state->get_automatic_gno(automatic_gtid.sidno);
    if (automatic_gtid.gno == -1)
    {
        gtid_state->unlock_sidno(automatic_gtid.sidno);
        return_reported_error;
    }
    gtid_state->acquire_ownership(thd, automatic_gtid);
    gtid_state->unlock_sidno(automatic_gtid.sidno);
}
```

修改之后的代码如下。

```

#ifdef WITH_WSREP
if (WSREP(thd) && thd->wsrep_trx_meta.gtid.seqno != -1)
{
    /* 在修改之后, GTID前半部分使用的还是Galera层的GTID字符串 */
    automatic_gtid.sidno= wsrep_sidno;
    /* 但是SESSION.GTID_NEXT值的后半部分, 就摇身一变, 成了galera_seqno值,
       也就是在Binlog内容中的Xid值。这样Xid值与SESSION.GTID_NEXT就是一样了 */
    automatic_gtid.gno=thd->wsrep_trx_meta.gtid.seqno;
}
else
{
#endif /* WITH_WSREP */
    /* 而只有在非Galera的情况下, SESSION.GTID_NEXT才保持与原来一样的值 */
    automatic_gtid.sidno= gtid_state->get_server_sidno();
    automatic_gtid.gno=gtid_state->get_automatic_gno(automatic_gtid.sidno);
#ifdef WITH_WSREP
}
#endif /* WITH_WSREP */
gtid_state->lock_sidno(automatic_gtid.sidno);
if (automatic_gtid.gno == -1)
{
    gtid_state->unlock_sidno(automatic_gtid.sidno);
    RETURN_REPORTED_ERROR;
}
gtid_state->acquire_ownership(thd, automatic_gtid);
gtid_state->unlock_sidno(automatic_gtid.sidno);

```

可以看到, 通过测试修改之后的代码, GTID 都是一致的。完整的 Binlog 内容如图 40.3 所示, 所示内容为上一个事务的 Commit 事件和下一个事务的开始事件。

```

#160518 13:22:04 server id 29199 end_log_pos 1377 CRC32 0x2830cb91 Xid = 907242391
COMMIT/*!*/;
# at 1377 the Xid and next gtid is same for one trx in all cluster nodes
#160518 13:22:04 server id 29199 end_log_pos 1425 CRC32 0x503e440b GTID [commit=yes]
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242392'/*!*/;next trx

```

图 40.3

截取 GTID_NEXT 事件和 Xid 信息, 如图 40.4 所示。

从图 40.4 中可以看到, 这些信息都是一致的了。但从上面的代码对照中可以看出, 在 PXC 中, 官方实际上是试图将 SESSION.GTID_NEXT 做一个修改, 但只改了前面一部分, 改得不

彻底。从本人与官方开发者的邮件往来中可以发现，他们认为这两部分完全相同并不会带来什么好处，所以只修改了前半部分。关于这个好处，可以通过下面这个需求来了解一下。

```
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242395'/*!*/;
#160518 13:22:04 server id 29199 end_log_pos 6121 CRC32 0x1bee3c9b Xid = 907242395
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242396'/*!*/;
#160518 13:22:04 server id 29199 end_log_pos 7307 CRC32 0x9b51c1a5 Xid = 907242396
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242397'/*!*/;
#160518 13:22:04 server id 29199 end_log_pos 8493 CRC32 0xa17b90e6 Xid = 907242397
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242398'/*!*/;
#160518 13:22:04 server id 29199 end_log_pos 9679 CRC32 0x018d59f5 Xid = 907242398
SET @@SESSION.GTID_NEXT= '75420d9e-1a3f-ee1a-66f2-b8d67fb74359:907242399'/*!*/;
#160518 13:22:04 server id 29199 end_log_pos 10865 CRC32 0x09697288 Xid = 907242399
```

图 40.4

在 Galera Cluster 中，Xid 是唯一的，通过这个值，可以唯一确认一个事务的执行情况。如果在 Galera Cluster 中的某个节点上挂有从库，那么该从库就可以根据这个 Xid 值来确认在不同节点之间来回切换的位置，这也是上面所叙述的第一种方法。相对来说，方法还算简单，但也实属被逼无奈，因为没有更好的方法了，这是唯一可靠的方法。

但是想想看，如果 SESSION.GTID_NEXT 也是相同的呢？这样的话，是不是就和 MySQL 5.6 版本中设计 GTID 的初衷可以对上了。并且更关键的是，SESSION.GTID_NEXT 值是在一个事务的开始位置，而 Xid 是在一个事务的结尾位置。可以再想想，如果在分析 Binlog 时，最前面就可以确认这个事务是否已经被执行过好呢？还是在分析到事务结尾时才发现这个事务是不是已经被执行过好呢？这是显而易见的，因为如果分析到结尾时才发现的话，那很明显，在分析这个事务的过程中，所有的事件都不能提前处理，需要先缓存起来，然后等到 Xid 时，才去确定这个事务是应该抛弃还是去做应用。那么设想一下，如果这个事务很大呢？我觉得这会非常苦恼。

这样的需求，现在已经非常多了，最常用的就是一些 Binlog 解析项目，分析完之后，将它们实时异构同步到其他数据库，比如 Oracle、Hbase 或 Redis 等。这种情况下，SESSION.GTID_NEXT 的集群统一就非常有意义了。因为有了它，就可以在集群中保证数据同步的完整性，这样就可以在当前同步主节点挂掉之后，根据 SESSION.GTID_NEXT 的值，在同一个集群中另一个节点的 Binlog 里，随便找一个比挂掉时的事件时间值早的位置开始复制，这样就可以通过 SESSION.GTID_NEXT 值来确认其执行情况，从而保证集群中的同步完整性。

可惜目前这只是一个想象，但我们相信，以后必然是这样的。本章内容是基于 Percona Xtradb Cluster (PXC) 的，目前版本中还没有支持这样的功能。

同时，MySQL 官方也出品了 Group Replication，不知道它会不会实现。关于 Group Replication 的具体讲解，请参考 27 章的相关章节。

总结

本章重点要讲的是，如何做到让一个从节点在 Galera Cluster 不同节点之间来回转移，所用到的特性就是 Xid 的值，即 Galera 的 GTID 值，关键的一点就是 Xid 集群中的不同节点之间，同一个事务对应的值是完全一样的，并且是有序的，所以只要拿到一个 Xid 值，就可以在其他节点中找到对应的位置及相关信息。

而和这个 Xid 具有同样设计思想的是 MySQL Server 的 GTID (SESSION.GTID_NEXT) 值。它要保证通过主从复制实现的集群中，同一个事务有一个共同的标签，即 GTID 值，这样就可以在集群中确保事务的唯一性了。

二者的设计理念如出一辙，但不幸的是行为却背道而驰，或者说二者目前还在两条平行的高速公路上飞驰着，如果有一天……那该多好啊！

41

Galera Cluster 节点与其从库的 随意转换

背景

在我们的业务中，有一个数据库集群采用的是 Percona Xtradb Cluster (PXC) 结构。应用部署在一个机房 (IDC1)，数据库部署在另一个机房 (IDC2)，由于机房间的网络问题，造成了很严重的数据延迟，所以打算在下一个高峰期之前把数据库和业务迁移到一起。当然，根据 DBA 的传统是尽量不给业务添麻烦，所以要默默地把数据库迁移到 IDC1 去。

此业务数据库有 1.2TB 数据，由于这个数据量偏大，在不同机房间迁移导致了工作量成倍的增加。这样的问题，即使是一个久经沙场的 DBA，心里也不是滋味——因为又要熬夜了。

遇到这样的问题，按照以往传统的做法，有以下两种方案。

- 采用传统的方式，选一个好日子，一般是凌晨两点半，大家无法入眠，手里握着计划和方案（嗯，不是卡拉 OK），找到所有相关 DBA、应用及 QA 相关人员，起来做 SST。在现在的集群中，新加入三个节点变成六个节点。新的三个节点都在 IDC1，这样在集群管理平台上，通过鼠标单击切换就可以搞定了。不过，说起来容易，对于 1.2T 的数据，SST 实际要做多久呢？简单算一下，大约需要 $1.2T * 1024 * 1024 / 3600s / 80MB = 4.4$ 小时。从一点半开始，4.4 个小时之后，就是六点，这时业务已经要处于高峰期了，所以做完一个就可以休息了。第一，身体已经受不了；第二，业务也受不了，所以今天就不做了。连

续三天,如果有一天失败了,那么连续 N 天,相关人员 OVER,噩梦!传统方式不靠谱,属于 DBA 噩梦型。

- 采用暴力方式,提前在 PXC 上加一个从库,或者是一个从 PXC 集群。中间是靠主从复制的方式同步数据,这些可以提前准备好。但是,这种方式给应用带来了很大的风险,就是需要挂免战牌,停服务。因为是主从方式,没法双写,读可以先切到从集群中,但写必须是全部在主集群中停掉之后,断掉中间的复制,然后再齐刷刷地改到从集群中。这种方式不会那么累,但是需要应用停机,对用户也不太友好,属于 DBA 友好型。

想到这里,到底选择哪种呢?突然有一种新的想法,难道暴力方式就没有办法变得优雅么?从库和 PXC 节点难道都是不共戴天的么?从库能转到 PXC 节点么?好像可以。

那就开始吧!先看看上面的暴力方式中使用的主从集群复制的架构图,如图 41.1 所示。

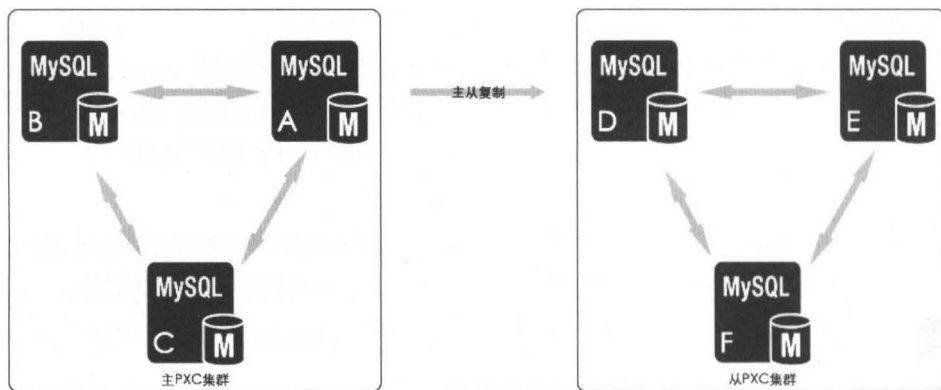


图 41.1

从节点向 PXC 节点的转换

下面要讲述的方法,是如何将一个 PXC 集群中某个节点的从库节点,在线快速地通过 IST (而不是 SST) 的方式加入到集群中。这样在运维上给 DBA 省去了很多熬夜的机会,具体操作步骤如下。

1. 最好保证在没有数据库复制延迟的情况下,将复制断掉,这样两个集群就是独立的了。(如图 41.4 所示,从集群 (D、E 和 F) 的三个节点,可以都挂在 A 节点下面的三个独立从库,也可以以 D 为主库, E、F 为挂在 D 节点下面的从库,总之 D、E 和 F 三个节点作为 A 节点的下游节点即可。下面的讲述都是建立在假设从集群也是 PXC 集群的前提下,即建立在图 41.4 的基础上)

2. 如果这个在 A 节点下游的三个从节点 (D、E 和 F) 是简单的 MySQL 从库, 而不是 PXC 节点, 就先将这个节点的数据库换成 PXC, 并且准备好相应的配置文件等。
3. 检查主 PXC 集群中的 A 节点, 记录其参数 `wsrep_local_cached_downto` 的值, 这个值是当前 GCache 中最小的 GTID 值, 用来查看从库最后执行的 GTID 是否比这个值还小, 如果是的话, 就不能做 IST 了。这里主要是为了检验。
4. 检查从集群中 D 节点的 relay-log 是否已经执行完, 如果是, 则把数据库实例 STOP 掉。D 节点停掉之后, 找到这个实例 Binlog 目录中的最后一个 relay-log 文件, 通过 `mysqlbinlog` 工具, 找到最后一个事务对应的 Xid 值, 那么这个值就是当前 D 节点执行的最后一个 A 节点上传过来的事务对应的 Xid 值。(这里最好校对一下, 已经执行的 Binlog 文件与 relay-log 文件的最后一个事务是否是相同的内容, 以防止在 relay-log 文件中找到的是一个还没有执行的事务, 注意这里对比的是内容, 而不是 Xid, 因为 Xid 是独立的, 一般是不同的)。
5. 修改 D 节点的配置文件, 包括 `my.cnf` 参数 (这是基本的), 还包括 `wsrep.cnf` 中的参数 `wsrep_cluster_name`, 修改为主 PXC 集群对应的参数。修改参数 `wsrep_cluster_address` 为主 PXC 集群某一个节点的地址, 修改参数 `wsrep_sst_donor` 为主 PXC 集群的某一个读节点 (尽量减少对线上的影响)。
6. 修改 D 节点 `data` 目录下的文件 `grastate.dat`, 将 `uuid` 修改为主 PXC 集群中的 `wsrep_cluster_state_uuid` 参数的值, `seqno` 修改为上面从 relay-log 中取到的最后一个 Xid 的值。
7. 万事俱备, 只差启动了。执行命令启动数据库并观察。启动成功, 并且成功加入, 如果操作迅速, 几分钟或几秒钟就可以同步完成。
8. 那么对于 E 节点和 F 节点来说, 操作过程是一样的, 重复执行两遍就搞定了。
9. 将 D、E、F 节点加入之后, 主 PXC 集群和从 PXC 集群就成为一个集群了。这样的数据库迁移方式其实就是最想要的那种方式, 只需要单击就可以搞定了, 好像十分钟就可以睡觉了, 再也不需要等几个晚上了。

说白了, 这种迁移方式是上面所说的两种方式的结合, 使用暴力方式准备数据, 使用传统方式进行迁移, 把传统方式中最费时间的在线工作转换为离线方式, 并且在任何时段做都不会影响业务, 把在线的工作精简到只需要在集群管理平台中操作上下线的问题, 这样很好地解决了迁移困难的问题, 并且极大地减小了 DBA 工作的压力, 值得推广。



注意: 在上面的第 4 点中, 一定要在 Relay-log 文件中找到最后一个 Xid。因为只有 Relay-log 才是原汁原味的主库 Binlog, 才能找到最后执行的 Xid (GTID) 值。因为通过主从异步复制同步过来的事务, 在 Binlog 文件中的 Xid 值是按照从库自己的事务 ID 来计算的, 所以有可能 (99.9% 的情况下) 是不同的。

PXC 节点向异步从节点的转换

上面已经介绍了从库向 PXC 转换的步骤及原理。假如有一个 PXC，要将某一个节点拿出来作为从库，那么要怎么操作呢？其实在了解了架构及同步位置的相关知识点之后，这个操作就变得非常简单了，具体操作步骤如下。

1. 停掉某一个要作为从库的 PXC 节点。
2. 到相应节点的 Binlog 目录下找到最后一个执行的 Binlog 文件，同样找到最后一个事务的 Xid 值。
3. 再到其主库节点的 Binlog 文件中找到这个 Xid 的位置（具体的查找方法，可以无所不用其极，因为不确定这个 Xid 在哪个文件中，所以可以使用 Binlog 的时间先确定是哪个文件，然后再在这个文件中找具体位置），然后找到对应事务的下一个事务的 Binlog 开始位置。那么这个位置就是 CHANGE MASTER 命令中要指定的位置。
4. 然后加复制权限。
5. 最后执行 CHANGE MASTER 命令即可。

这种方式实际上和之前写的一个关于从库如何在不同的 Galera Cluster 节点之间转移的操作方法类似，主要是确定数据同步的位置。此外也要明白一点，在 PXC 中，集群中完全同步的只有 Xid 值，我们做这些转换，都要依赖这个值，才能在不同节点之间过渡（不过有一个前提，就是假设参数 `repl.commit_order` 的值为 3）。

至此，我们已经知道，PXC 节点与从库节点的随便切换变得非常容易。这种方式真的是给 PXC 迁移带来了福音，同时也给 DBA 带来了福音！

最后，可以多睡几个好觉了！

业务更新慢，不是由 Galera 引起的

在做从 MMM 到 PXC 集群的常规数据库迁移的时候，发生过一件有趣的事情。这个迁移过程，我们熟悉得不能再熟悉了，因为这是在最近两年内一直做的一件事。我们很相信 PXC，不管是稳定性、性能，还是对数据的一致性，亦或是 DBA 运维的难易程度，都是备受推崇的，所以会尽量把所有重要的业务都迁移到 PXC 上面，而此案例中的数据库迁移便是其中的一个。

迁移时的架构图如图 42.1 所示。

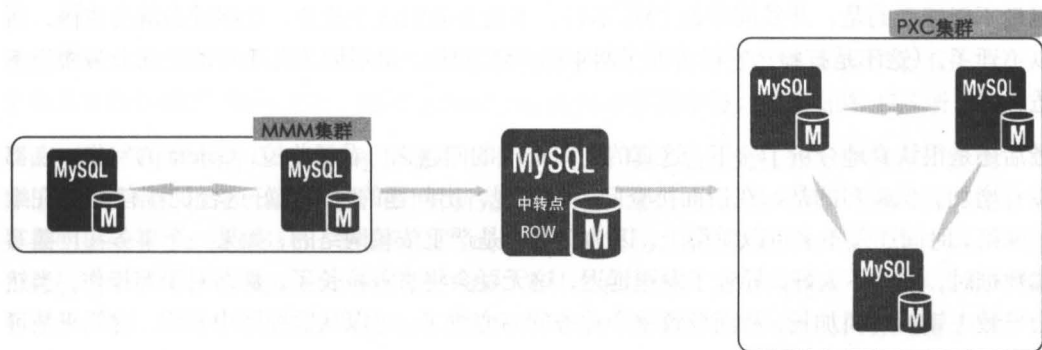


图 42.1

如图 42.1 所示，是从一个 MMM 集群迁移到一个 PXC 集群的过程，但中间多出来一个中转的节点。这个节点在迁移过程中是需要的，因为 PXC 需要的 Binlog_format 是 ROW 格式的，

而 MMM 集群的 Binlog_format 可能是 STATEMENT 或 MIXED 格式, 所以需要一个中间节点, 将格式转换为 ROW。这便是这个中转点的作用。如果没有这个转换的话, 有时候会因为自增 ID 的问题导致数据出错。因为 PXC 的自增 ID 步长和 MMM 是不一样的, 所以这样做会保险很多。

迁移过程就不多说了。但是迁移完成之后, 开发同学反映更新变得越来越慢, 可以从图 42.2 中的对比看出。



图 42.2

很明显可以看出, 这种影响已经到了没办法接受的地步了。在发现的时候, 没有多少时间考虑具体的解决方案, 所以就只能先做回滚了。值得庆幸的是, 这个被影响的业务只有更新语句, 并且是只针对一个表的更新, 只需要保证最新的数据能写成功, 而不必保证当前被写的数据是否是最新的, 如果有数据不正确, 也可以手动更新恢复。同时, 由于表的数据量非常小, 只有 3000 行, 所以很容易就回滚回去了。但不好是, 现场没有了, 只留下了几张监控图, 这个可以用于后面的分析。

但最不能接受的是, 开发同学说 PXC 不行, 不适合他们这个业务, 迁移过去就会变慢, 所以不迁了。(这不是打脸么? 极力推了两年的 PXC 架构, 最后因为几千行的更新业务慢说不适合。)

最后还是很认真地分析了一下, 这真的是 Galera 的问题么? 看了监控, Galera 的写集一点都没有增加。但碰巧的是, 在后面排查问题时发现, 出问题的时候正好这台机器有 OPS 在维护网络, 时间还差不多可以对得上。因为 Galera 是严重依赖网络的, 如果一个事务在广播写集数据时, 网络不太好, 导致了发送延迟, 这无疑会将事务拉长了。那么对于写操作, 当然会导致上锁的时间加长, 从而导致整个业务的写变慢了。可以从监控图中看到, 这似乎是可以对得上的, 如图 42.3 所示。

从图 42.3 中可以看出, 出问题的时间段, 锁等待时间确实上升了, 同时, 从图 42.4 所示的 Processlist 的监控上也可以看出这一点。

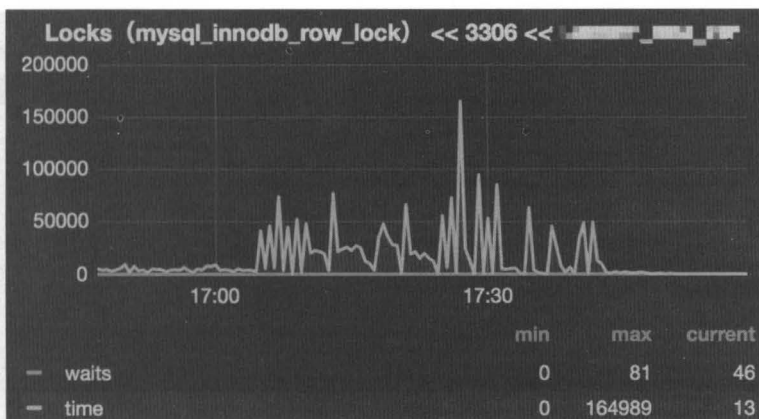


图 42.3

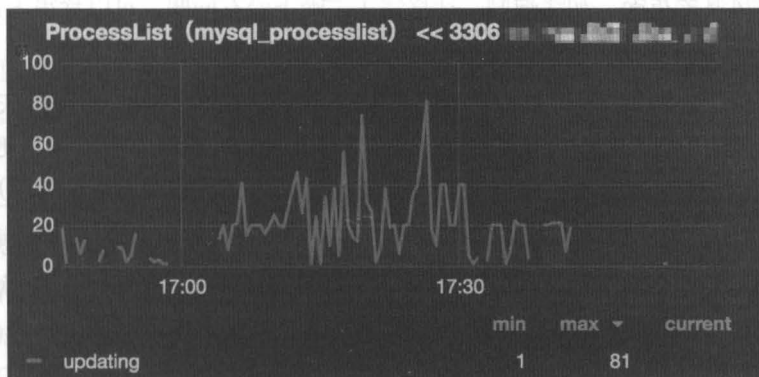


图 42.4

关于排查大事务的问题，还有一个方法就是直接问开发同学：“这个更新操作是不是放在一个事务中执行的？”确认之后，通过 `general_log` 找到对应的事务，发现如图 42.5 所示。

从图 42.5 中所展示的 General log 信息可以看出，操作顺序是，先打开一个事务，查找了很多信息，然后拼出来一条更新语句，更新完之后再查一次，加上隔离级别是 RR 的，这个事务很明显就长了。经过分析，感觉可以写成一条更新语句，所以希望和开发同学商量一下能不能把这个改掉。结果开发同学回了一句：“为什么在 MMM 上没问题？”我表示无话可说，那不改了吧。（自己也觉得的确是这样。为什么同样的语句原来没问题现在有问题了呢？）

但此时没有了现场，排查问题总是有心无力，可是这个问题不能就这样过去了。如鲠在喉，总感觉有什么事，就这样让迁移中止了，以后还怎么再推 PXC 上线。由于业务很重要，我们本着谨慎的态度和开发同学进行了深刻的问题分析，并耐心地讲道理，终于同意找时间再上线一次。但也不能确定是网络原因导致的，虽然时间是能够对得上。

```

101219558 Query SET autocommit=0
101219558 Query SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ
101219558 Query select id, groupInt, nodeId, ip, port, nodeId_md5 from snode where groupInt
= 15 and status = 1
101219558 Query select id, groupInt, nodeId, ip, port, nodeId_md5, status, f_usage, quota f
rom snode where id in(407348066,407348065,407348064,407348063,407348060,407348062,407348061,407348055,40734
8054,407348052,407348050,407348047,407348048,407348053,407348049,407348046,407348051,407348045,407348044,40
7348025,407348026,407348022,407348017,407348013,407348014,407348011,407348012,407347996,407347999,407348000
,407347997,407347995,407347994,407347998,407347993,407347967,407347970)
101219558 Query UPDATE snode SET status = 2 , machine_mark= 4, uCount = uCount + 1, f_usage
=f_usage + 1 WHERE status = 1 and id in (407348012,407348013,407348014,407348022,407348025,407348026,407348
044,407348045,407348046,407348047,407348048,407348049,407348050,407348051,407348052,407348053,407348054,407
348055,407348060,407348061)
101219558 Query select id, groupInt, nodeId, ip, port, nodeId_md5, status, f_usage, quota f
rom snode where machine_mark= 4 and id in (407348012,407348013,407348014,407348022,407348025,407348026,4073
48044,407348045,407348046,407348047,407348048,407348049,407348050,407348051,407348052,407348053,407348054,4
07348055,407348060,407348061)
101219558 Query commit

```

图 42.5

又是一个夜黑风高的夜晚，如法炮制。迁移完了，晚上没有问题，可以称得上是“平安夜”了。

等到第二天下午，感觉业务跑得挺好，没什么问题，难道真的是网络问题？昨天晚上迁移的业务只是出问题的那部分写，还有一个业务是读业务，只读固定的数据，不会更新，所以没有迁移过来。下午继续迁移（为什么要选择下午，之前出问题的时间也是下午）。

迁移读，QPS 很小，是另一个业务。读取的是完全不同的表，迁移过去了，想着这事就算过去了。过了几十分钟，业务同学突然说，相同的问题又出现了，时间和前一次差不多（这是巧合还是巧合？）。下面集中火力查找这个问题，看了一下 show engine innodb status，发现了如图 42.6 所示的这些问题。

```

---TRANSACTION 636120540, ACTIVE 1 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1184, 2 row lock(s)
MySQL thread id 4537284, OS thread handle 0x7fbafeab4700, query id 1657508502 clientip allianceRouter up
dating
update snode set status = 1, uCount = uCount + 0 , f_usage = f_usage + 0, sub_groupInt = 0 where nodeId
= 'G53.15718080#36' and status = 2
----- TRX HAS BEEN WAITING 1 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 2047 page no 55 n bits 192 index `PRIMARY` of table `app_database`.`snode` trx id
636120540 lock_mode X locks rec but not gap waiting
-----
TABLE LOCK table `app_database`.`snode` trx id 636120540 lock mode IX
RECORD LOCKS space id 2047 page no 57 n bits 512 index `uniq_key_nodeId` of table `app_database`.`snode`
trx id 636120540
lock_mode X locks rec but not gap

```

图 42.6

满屏的锁等待，更新业务肯定会慢。由于时间紧迫，起初以为是 5.5 到 5.6 版本的执行计划变化（优化），有可能导致在 5.6 版本中的相同语句，有的走的是正确的索引，有的走的是错误的索引，所以导致了同一条语句交叉更新同一行数据（问过开发同学，说存在这种情况）。

这样的话，无疑就造成了很多锁冲突，使更新变慢。但此时发现，这条语句，与之前所说的在开启了显式事务中先查后更新的不是同一条语句。所以，继续在 engine status 中找。最后，在最下面找到了如图 42.7 所示的信息。

```

---TRANSACTION 636119883, ACTIVE 1 sec
MySQL thread id 4539761, OS thread handle 0x7fbfd92f700, query id 1657506819 clientip allianceRouter
cleaning up
Trx read view will not see trx with id >= 636119884, sees < 632758957
---TRANSACTION 636119677, ACTIVE 1 sec
3 lock struct(s), heap size 1184, 20 row lock(s)
MySQL thread id 4536468, OS thread handle 0x7fbf6142a700, query id 1657506739 clinetip allianceRouter
cleaning up
Trx read view will not see trx with id >= 636119678, sees < 632758957
TABLE LOCK table `app_database`.`snode` trx id 636119677 lock mode IX
RECORD LOCKS space id 2047 page no 55 n bits 192 index `PRIMARY` of table `app_database`.`snode` trx
id 636119677 lock_mode X locks rec but not gap
RECORD LOCKS space id 2047 page no 56 n bits 192 index `PRIMARY` of table `app_database`.`snode` trx
id 636119677 lock_mode X locks rec but not gap

```

图 42.7

这才恍然大悟！原来不是同一条语句出现了交叉锁等待，而是与之前提到的显式事务出现了交叉锁等待。也就是说，那个显式的事务，每次先查后更新 20 行数据，而另一个单独自动提交的事务（从 general_log 中看到是自动提交的）会更新同样的数据，而自动提交使用的索引是 `uniq_key_nodeId` 的唯一索引，显式事务使用的是主键索引，更新时每一个索引都会被更新，所以导致了交叉锁等待。

似乎问题已经清楚了，交叉锁等待肯定会慢，所以还是找开发同学问一下能不能紧急改掉，把显式事务改成一条更新语句，这样事务执行很短暂，问题肯定可以解决。但开发同学很坚定：“为什么在 MMM 上没有问题？”（因为他们的理由很简单，MMM 已经用了五年，基本都没问题，一上 PXC 就有问题，那还不是 PXC 的问题么？）我又无话可说了。

不能通过改代码来解决，就只能继续排查问题了。但时间不多，现在延迟已经非常严重，如何缓解呢？是不是机器负载有点高，导致事务拉长比较明显呢？所以这个问题就凸显出来了。通过切换，将写流量都打到另一个节点上，确实有效，延迟没有了。但问题还没有查清楚，继续查吧（也就是用这种方式，给了更多的缓冲时间）。

此时注意到，之前的那个问题还是出现了，UNDO LOG 的未 PURGE 回滚段还是一直在涨，还是那个坚定的解释——有大事务。

此时再次分析 Binlog，由于是 ROW 格式的，分析起来比较困难，因此写了一个脚本，可谓是利器，脚本内容如下。

```
#!/bin/bash
```

```
BINLOG_FILE="mysql-bin.000046"
```



```

START_TIME="2015-06-28 8:45:00"
STOP_TIME="2015-06-28 10:10:00"
mysqlbinlog --base64-output=decode-rows -vv --start-datetime="${START_TIME}"
--stop-datetime="${STOP_TIME}" ${BINLOG_FILE} | awk \
'BEGIN {xid="null";s_type=""; stm="";endtm="";intsta=0;inttal=0;s_count=0;count=0;
insert_count=0;update_count=0;delete_count=0;flag=0;bf=0;period=0;} \
{
if (match($0, /^(BEGIN/)) {bg=1;} \
if (match($0, /#.*server id/)) {if(bg==1){statm=substr($1,2,6)" "$2;
cmd=sprintf("date -d \"%s\" +%s", statm);cmd|getline intsta;close(cmd);bg=0;bf=1;
}else if(bf==1){endtm=substr($1,2,6)" "$2;cmd=sprintf("date -d \"%s\" +%s", endtm);
cmd|getline inttal;close(cmd);}} \
if(match($0, /#.*Table_map:.mapped to number/)) {
printf "Timestamp : " $1 " " $2 " Table : " $(NF-4); flag=1} \
else if (match($0, /#.*Xid =.*/)) {xid=$(NF)} \
else if (match($0, /(### INSERT INTO *.*))/) {count=count+1;
insert_count=insert_count+1;s_type="INSERT"; s_count=s_count+1;} \
else if (match($0, /(### UPDATE *.*))/) {count=count+1;
update_count=update_count+1;s_type="UPDATE"; s_count=s_count+1;} \
else if (match($0, /(### DELETE FROM *.*))/) {count=count+1;
delete_count=delete_count+1;s_type="DELETE"; s_count=s_count+1;} \
else if (match($0, /^(# at) /) && flag==1 && s_count>0) {
print " Query Type : " s_type " " s_count " row(s) affected" ;s_type=""; s_count=0; }
\
else if (match($0, /^(COMMIT/))){period=inttal-intsta;if(inttal==0){period=0};
print "[Transaction total : " count " Insert(s) : " insert_count
" Update(s) : " update_count " Delete(s) : " \
delete_count " Xid : "xid" period : "period" ] \n+-----+
-----+-----+"; \
count=0;insert_count=0;update_count=0; delete_count=0;s_type="";
s_count=0; flag=0;bf=0;bg=0;} } '

```

做的事情比较多，所以分析起来比较慢。在用的时候，只需要修改要被分析的文件名，以及需要的时间段即可。分析结果如图 42.8 所示。

分析结果的信息非常丰富，可以找到一个事务中修改了多少行，用于查看有没有更新（插入、删除）非常多的事务。这也是大事务的一种，当然还可以看是在什么时间做的操作，这里最重要的就是可以找到一个事务的执行时间——period，以秒为单位。有了这些信息，要找一个大事务就非常容易了。分析结果如图 42.9 所示。

```

+-----+
Timestamp : #161103 14:58:55 Table : `app_database`.`snode` Query Type : UPDATE 1 row(s) affected
[Transaction total : 1 Insert(s) : 0 Update(s) : 1 Delete(s) : 0 Xid : 85115583 period : 0 ]
+-----+
Timestamp : #161103 14:58:55 Table : `app_database`.`snode` Query Type : UPDATE 1 row(s) affected
[Transaction total : 1 Insert(s) : 0 Update(s) : 1 Delete(s) : 0 Xid : 85115584 period : 0 ]
+-----+
Timestamp : #161103 14:58:55 Table : `app_database`.`snode` Query Type : UPDATE 20 row(s) affected
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 85115585 period : 1 ]
+-----+
Timestamp : #161103 14:58:56 Table : `app_database`.`snode` Query Type : UPDATE 1 row(s) affected
[Transaction total : 1 Insert(s) : 0 Update(s) : 1 Delete(s) : 0 Xid : 85115586 period : 0 ]
+-----+
Timestamp : #161103 14:58:56 Table : `app_database`.`snode` Query Type : UPDATE 1 row(s) affected
[Transaction total : 1 Insert(s) : 0 Update(s) : 1 Delete(s) : 0 Xid : 85115587 period : 0 ]
+-----+

```

图 42.8

```

# cat sum|grep Transaction|awk '{if($19>0)print}'
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84371744 period : 1 ]
[Transaction total : 5 Insert(s) : 0 Update(s) : 5 Delete(s) : 0 Xid : 84440913 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84449513 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84515676 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84524371 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84577924 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84649406 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84670213 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84682259 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84685121 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84695077 period : 1 ]
[Transaction total : 20 Insert(s) : 0 Update(s) : 20 Delete(s) : 0 Xid : 84713198 period : 1 ]

```

图 42.9

很神奇地发现，最长的事务就是 1 秒钟，被影响的数据行数，最多就是 20 行。这个 20 行与上面 show engine innodb status\G 中看到的 20 行行锁能够对上，通过查看库表信息可以证实确实是同一个事务。那么也就是说，显式的事务更新时间用了 1 秒，一个事务 1 秒，并且锁了 20 行，这必然会慢。

但还是那个问题，为什么在 MMM 上没问题，在 PXC 上就有问题呢？百思不得其解。只能再次从回滚段未 PURGE 的量入手。继续看 innodb status，内容很长，重点关注参数 Innodb_history_list_length，如下。先观察一段时间 Innodb_history_list_length 的变化，只增不减，与监控图是一致的。

```
mysql> show status like "%hist%";
```

```

+-----+
| Variable_name | Value |
+-----+
| Innodb_history_list_length | 840501 |
+-----+

```



```
1 row in set (0.00 sec)

mysql> show status like "%hist%";
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| InnoDB_history_list_length | 840584 |
+-----+-----+
1 row in set (0.00 sec)
```

同时,继续关注 innodb status 的 TRANSACTION 部分,如图 42.10 所示。

```
-----
TRANSACTIONS
-----
Trx id counter 217863927
Purge done for trx's n:o < 213094702 undo n:o < 0 state: running but idle
History list length 846462
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started
MySQL thread id 257580, OS thread handle 0x7f4383451700, query id 127.0.0.1          init
show engine innodb status
---TRANSACTION 217863354, not started
MySQL thread id 257581, OS thread handle 0x7f4381f7f700, query id 305128717
cleaning up
---TRANSACTION 217861947, not started
MySQL thread id 257549, OS thread handle 0x7f3fa6515700, query id 305123105
cleaning up
---TRANSACTION 217863256, not started
MySQL thread id 257497, OS thread handle 0x7f438a92e700, query id 305126827
cleaning up
```

图 42.10

可以发现有很多事务都是未开始状态,但事务多版本中最大可见事务号一直保持 213094702 不变。突然想到,难道存在至今还长期未提交的事务吗?所以导致最早的事务一直不能 PURGE?但关键是,出问题的事务都提交了(不管从 Binlog 还是从 general_log 中都可以来证明这一点),怎么会有没提交的事务。难道有读事务,由于 innodb status 的内容太长了,忽略了什么重要的信息?接着一行一行地往下看,终于看到了下面的内容(如图 42.11 所示),一切就都了然了。

在内容的最下面,看到了原来有 3700 秒没有提交的 ACTION 事务,应该是这个问题了。

快速浏览这些内容,发现对应的 thread_id,然后在 general_log 文件中找到了对应的语句,如图 42.12 所示。

这是哪里来的业务表呢?查了一下,正是下午发布的业务,就是那部分不会影响业务的读。为什么不在上午发呢,这样不就和上次区分开了?难道是与这个读业务的发布有关系?

```

---TRANSACTION 213136590, ACTIVE 3704 sec
MySQL thread id 256041, OS thread handle 0x7fcb4d7e1700, query id 307450110      cleaning up
Trx read view will not see trx with id >= 213136591, sees < 213094700
---TRANSACTION 213128600, ACTIVE 3708 sec
MySQL thread id 256402, OS thread handle 0x7fcc8bfbef00, query id 307448794      cleaning up
Trx read view will not see trx with id >= 213128601, sees < 213094700
---TRANSACTION 213125667, ACTIVE 3709 sec
MySQL thread id 256389, OS thread handle 0x7fcb55df9700, query id 307447305      cleaning up
Trx read view will not see trx with id >= 213125668, sees < 213094700
---TRANSACTION 213124885, ACTIVE 3710 sec
MySQL thread id 256031, OS thread handle 0x7fc66fc9f700, query id 307403674      cleaning up
Trx read view will not see trx with id >= 213124886, sees < 213094700
---TRANSACTION 213124875, ACTIVE 3710 sec
MySQL thread id 256042, OS thread handle 0x7fc99e79f700, query id 307450106      cleaning up
Trx read view will not see trx with id >= 213124876, sees < 213094700

```

图 42.11

```

256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='MLE' AND rrr='PEK' AND sampleline ='EY'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='PEK' AND rrr='TPE' AND sampleline ='CI'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='SPK' AND rrr='TYO' AND sampleline ='NH'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='TYO' AND rrr='SHA' AND sampleline ='JL'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='PEK' AND rrr='NYC' AND sampleline ='QR'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='PEK' AND rrr='JKT' AND sampleline ='SQ'
256375 Query      SELECT 1
256375 Query      select cmd_type from apptable where ddp='CAN' AND rrr='KUL' AND sampleline ='MF'

```

图 42.12

快速找到对应的开发同学，让他们赶紧把这部分读业务回滚掉，几分钟之后，看到监控图如图 42.13 所示。

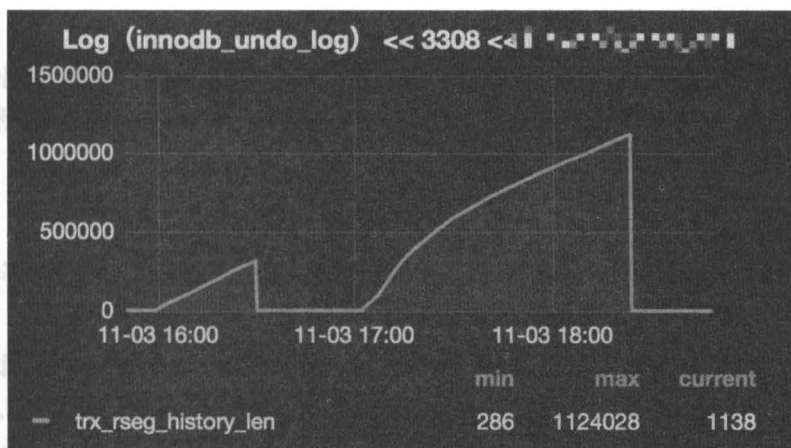


图 42.13

应用程序的加载时间如图 42.14 所示。

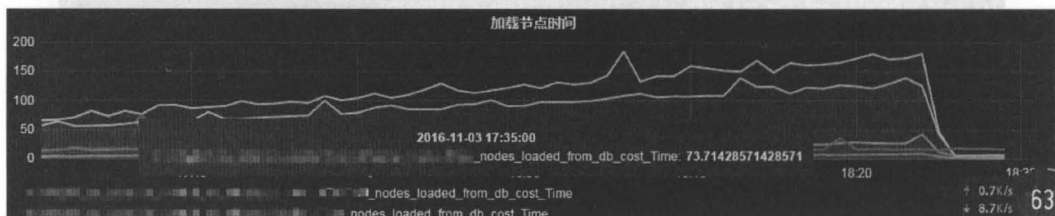


图 42.14

程序恢复了，读业务回滚了，那怎么会出现未提交的读事务呢？通过与开发同学的沟通发现，原来他们使用的 PXC 客户端是老版本的客户端。这个版本中的参数 `auto_commit` 默认为 `FALSE`，而新版本改为了 `TRUE`。这个点在迁移时沟通过，但可能被忽略了，没有落实到位。而在原来的 MMM 集群的连接上，默认都是 `TRUE`，这就能解释得通“为什么在 MMM 上没问题，而在 PXC 上是有问题的”了。

之后，按部就班，升级客户端，上线，之后就没有这样的问题出现了。

不过回过头来，还是有一个问题值得改进，那就是两个事务的交叉上锁。应该赶紧改成一个自动提交的更新语句。也许某一天，由于其他原因导致机器或数据库性能低的时候，这个问题还会再次暴露出来。这次是通过这种方式暴露出来的，那么下次就不好说了。所以开发同学经常会说：

- 为什么之前跑着没问题？
- 没有特别明确的理由，程序不可能改。
- MMM 跑了这么多年了，一直没问题，为什么要迁到 PXC？
- MMM 用着挺好的，优先级特别低。

我想说的是，DBA 通过让开发同学改代码来改进、优化，确实困难。驱动力也许就是在网络不稳定的时候，由于 VIP 误切导致数据乱了，开发同学自己修两个月的数据，然后就会主动了解 PXC 的优势并积极响应迁移到 PXC 上面了。

总结如下。

- 由于出现的问题所涉及的真实表与问题本身没有关系，导致没有快速找到问题的核心原因。
- 由于第二次读业务发布与第一次出问题的时间点是一致的，从而联想到是所谓的业务高峰期导致的，而没有想到是由发布本身引起的，从而转移了问题排查的方向。
- 由于一开始就考虑过 MMM 和 PXC 的不同，查找过配置文件和连接串，但都没有注意到默认参数 `auto_commit` 的问题。

- 一开始就知道，参数 `Innodb_history_list_length` 一直在涨的原因是存在没有提交的事务，而定向思维导致把更多的精力放在了写事务上面，而不是读事务，从而也导致查问题的方向被转移了。因为一般情况下，读事务是不会上锁的，不会导致从表象上看到的行锁等问题。但是读事务在 RR 隔离的级别下，影响还是很大的，这一点需要多多注意。
- `show engine innodb status` 内容太多，容易忽略很多细节的东西，还是需要培养取其精华，去其糟粕的能力。
- 在 `show engine innodb status` 信息中，如果有这方面的监控或巡检的话，可以在长事务上加上监控，比如超过 100 秒就可以体现出来，这样可以快速地定位问题。
- 在业务响应时间非常长的时候，切换一下可以缓解问题的原因，即在一个新的节点上，由于读事务未提交而导致回滚段不能释放的问题还不严重。随着时间的推移，事务不断地积累，回滚段中未 PURGE 的事务就会越来越多，从而导致一个事务的完成会非常慢，所以就越来越明显，需要在不断的切换中实现缓兵之计。
- 在读事务没有提交的情况下，从 `Processlist` 中是看不出来的，都是 `sleep` 状态，所以这也是导致查找问题不太直截了当的原因之一。
- 慢不一定是 PXC (Galera Cluster) 的问题，请放心使用。
- 可以将默认隔离级别都改为 `Read-Committed`，这样这种问题就不会出现，并且可以避免很多死锁问题。当然改为这个参数值的同时，还要把 `Binlog_format` 改为 `ROW`，不然会出现复制的问题。

43

在线改表引发的 Galera Cluster 集群死锁

背景

说起来，发现这个问题也算是巧合，正应了古人的一句名言：有意栽花花不开，无心插柳柳成荫。这件事说来话长，其实主旨也不完全算是构造死锁，而是涉及 Galera 及 MySQL 源码中具有争议的 Bug。那么就借这个主题，将导致死锁涉及的点及过程来讲清楚。

因为一直致力于 MySQL 的一些周边工具的开发及研究，并且努力在目前已经非常流行的 Inception 上面加一些强大的、让人眼前一亮的功能，比如在线改表，所以最初是想以一种全新的方案实现另一个在线改表的工具。目前比较流行的在线改表工具是 Percona 出品的 pt-online-schema-change，已经被集成到了 Inception 中，并且得到广泛使用。

而对于 pt-online-schema-change 存在的问题，相信每一位业内人士也都心知肚明，主要包括以下六点。

- 在 MySQL 中的触发器，是解释型的，不会做一些编译操作。每次触发都会导致当前事务产生额外的写入，并且这个写入操作的性能不可预知。同时，对触发器进行解析操作的性能也不够好。这样会极大地降低在线业务的性能，特别是对频繁使用的表而言。
- pt-online-schema-change 在改表的过程中，需要创建三个触发器，分别是 INSERT、UPDATE 及 DELETE 触发器。那么相对应地，触发器在使用完后还需要删除，删除操作又

会有三次。这样涉及的触发器操作总共会有 6 次。而我们知道，在 MySQL 表上，建立及删除触发器是需要对表加锁的，这样无疑给数据库的业务运行带来了不少压力。更重要的是，针对 Galera Cluster 而言，之前提到过，对表加锁会使得 Galera Cluster 杀死所有正在使用这个表的事务。这样造成的后果是，在线业务莫名出现了很多死锁。看过第 29 章的内容后，应该清楚在事务被杀死之后，都是报这个错误。那么，这个问题对于 pt-online-schema-change 而言，更是雪上加霜，这也就是它使用触发器直接带来的问题。

- 在 pt-online-schema-change 中，考虑到在改表过程中对数据库的影响，做了安全性的检查，比如连接数、并发数、数据库延迟等。如果发现这些指标有问题，就会暂停甚至中止改表的过程。这个想法是好的，并且很大程度上也避免了潜在的问题。但是，这个暂停只针对旧表到新表数据的搬迁过程，而其他方面的压力控制是没办法做的，包括触发器。也就是说，当数据库压力非常大的时候，只保证搬迁不做了，而在线业务打过来的增删改的流量同样还会继续触发触发器。只暂停了搬迁有可能根本缓解不了问题，而如果此时去中止改表，又会删除触发器，这三个动作可能就变成了压死骆驼的最后一根稻草。
- 通过使用 sysbench 测试，在使用 pt-online-schema-change 来改表时，触发器对当前表的 QPS 影响非常大，能降到非改表时的十分之一，这对于要求比较高的业务而言是无法忍受的。
- 另一个经常出现的问题是，pt-online-schema-change 工具创建的触发器，经常会和业务的写入请求发生死锁，从而影响业务的正常运转。
- 综上所述，如果是多表修改，就会存在更多的触发器。

所有上面提到的问题，导致在工具 pt-online-schema-change 中，触发器成为了最大的问题，也是经常被人们所诟病的问题。但苦于没有更好的工具，所以一直在使用它。

然而，想法是无穷的，创新也是无穷的。自从有了 Inception，就一直有一个想法，就是能不能把 pt-online-schema-change 工具的实现方式拆开、组装，放入到 Inception 中，把它变成一个原生的、与 Inception 集成在一起的功能，而不是在内部创建一个管道及进程调用 pt-online-schema-change 的功能。但如果是原原本本地搬过来，能有多少产出比呢？因为考虑到这个问题，所以一直没有下手。但还是想要做成这件事情，同时又解决 pt-online-schema-change 的一些问题，于是就诞生了本章所要讲的——想要做的一件事情。

用 Binlog 来代替触发器

pt-online-schema-change 中的触发器，说到底，就是为了处理在改表过程中的增量写入的。那么，增量写入的处理能不能使用 MySQL 界最著名的强大的 Binlog 呢？不知道在什么情况下，突然有了这么一个念头，差点就在 Inception 上实现了这个更伟大的功能，即轻量级的全能在线改表。

为什么说是差点呢？本来这个想法是好的，但受限于 MySQL 的 Bug，导致实现起来困难重重，从而引出了今天的问题。

使用 Binlog 来做增量，是一个好想法，但改表的总思路是不变的。首先，创建一个影子表。然后，找到老表及新表的主键，判断主键有没有被修改掉。如果主键被修改掉，则会导致改表之后的数据存在不正确的可能性。如果主键没有改掉，则通过主键的范围查询，将原表的数据做分片处理。一片一片地从原表里查出来，然后插入到新表中。不过这里需要注意的是，对于增删列而言，要确定在搬迁数据的过程中 SQL 语句要如何写。执行删列操作的时候，被删除的列既不能被查，也不能被写，因为这个列在新表中已经不存在了，这只是一个细节的举例。

这些内容与 pt-online-schema-change 的处理大同小异，而关键就是增量数据的处理。这里可以顺便说一下在 pt-online-schema-change 中的三个触发器的微妙之处，如下。

```
CREATE TRIGGER `pt_osc_db1_t1_del`
AFTER DELETE ON `db1`.`t1`
FOR EACH ROW
DELETE IGNORE FROM `db1`.`_t1_new`
WHERE `db1`.`_t1_new`.`id` <=> OLD.`id`;

CREATE TRIGGER `pt_osc_db1_t1_upd`
AFTER UPDATE ON `db1`.`t1`
FOR EACH ROW
REPLACE INTO `db1`.`_t1_new` (`id`, `name`) VALUES (NEW.`id`, NEW.`name`);

CREATE TRIGGER `pt_osc_db1_t1_ins`
AFTER INSERT ON `db1`.`t1`
FOR EACH ROW
REPLACE INTO `db1`.`_t1_new` (`id`, `name`) VALUES (NEW.`id`, NEW.`name`);
```

DELETE 触发器

在原表中的删除操作，对于每删除的一行，在新表中要执行的是 DELETE IGNORE FROM db1.t1_new WHERE db1.t1_new.id <=> OLD.id 操作。其中，关键在于关键字 IGNORE。因为在搬迁的过程中，删除数据的操作指不定要删除哪行，删除的有可能是已经搬迁过的数据，也有可能是还没有搬迁的数据。那么对于已经搬迁的数据，可以正常在新表中删除，而对于没有搬迁的数据，在新表中是找不到的，忽略即可。因为在后面搬迁到被删记录的位置时，这条记录在老表中已经被删除，也不会被搬迁，数据保持一致。

INSERT 触发器

在原表中的插入操作,对于每插入的一行,在新表中要执行的是 REPLACE INTO db1.t1_new (id, name) VALUES (NEW.id, NEW.name) 操作。这里的关键问题是 REPLACE INTO。想必很多人都会问:“一条插入语句,为什么触发的是 REPLACE INTO,而不是 INSERT 语句?”可能很多人会百思不得其解。其实,如果只是想着在老表做的只是简单的 INSERT INTO 语句,那么在从库中确实只需要执行对应的 INSERT INTO 语句即可。因为只要老表能插入成功,则新表也能插入成功,并且数据也是一致的。但是这里处理的问题,不是简简单单的 INSERT INTO 语句。因为在 MySQL 中,REPLACE INTO 语句也会触发 INSERT 触发器。可能讲到这里,大家就都明白了,这里的触发动作全是因为这种情况。那么新的问题产生了,REPLACE INTO 触发的为什么不是 UPDATE 触发器呢?在下面的“UPDATE 触发器”中会讲到其触发机制。到此可以发现,使用 REPLACE INTO 实际上是最合适的,能够兼顾到 INSERT INTO 语句和 REPLACE INTO 语句,可以很好地完成数据增量的处理工作。

UPDATE 触发器

在原表中的更新操作,对于每更新的一行,在新表中需要执行的操作是 REPLACE INTO db1.t1_new (id, name) VALUES (NEW.id, NEW.name)。可以看到,和 INSERT 触发器是一样的,并且很多人还会有疑问:“为什么触发的不是 UPDATE 语句呢?”如果是更新操作,并且产生了触发动作,那必须是已经存在的数据。对于新表,如果存在就更新,不存在就忽略。为什么不不存在的时候还要用 REPLACE INTO 让这条数据存在呢?其实这里的道理和上面的 INSERT 触发器是一样的。因为在 MySQL 中,不只是 UPDATE 语句会产生 UPDATE 的效果,有另一种叫 INSERT INTO ... ON DUPLICATE UPDATE 这样的语句,也会更新数据。而最最重要的是,这样的语句在 MySQL 中触发的竟然是更新触发器。只能说不按常规出牌了。那么这样的话,UPDATE 触发器长这个样子也就无可厚非了,也应该是这个样子的。

上面所讲的是 pt-online-schema-change 对于增量数据的处理方式,考虑周到,全方位,值得我们学习。但对于我们要做的,用 Binlog 来代替触发器,将会有什么变化呢?

换成 Binlog,首先要考虑的就是,必须基于 ROW 模式的 Binlog 才能做增量。因为这样才能准确找到被影响的数据,然后应用到新表中。

用 Binlog 来代替触发器,同样地,也分三种操作。

DELETE 事件

对于 DELETE 事件,既然产生了 Binlog,那么必然是在老表中就有这条数据。但是同样地,在新表中有可能存在,也有可能不存在。这和触发器方式没有区别,通过将 DELETE 事件转

换为一个 `DELETE IGNORE FROM db1._t1_new WHERE db1._t1_new.id <=> OLD.id` 的语句即可完成数据的一致性同步。

INSERT 事件

对于 INSERT 事件，在老表中执行的必然是 INSERT，并产生一条新记录，这与前面的触发器方式是有区别的。而如果是 REPLACE INTO，并且是已经存在的话，那就是 UPDATE 事件了。所以在这种情况下，对应的增量处理语句就是 INSERT IGNORE INTO 这样的语句。其实，这里的 IGNORE 也应该是多余的，因为在老表中可以插入成功，那么在新表中必然不会存在，所以肯定也可以正常插入。

UPDATE 事件

对于 UPDATE 事件，在老表上必然是实实在在地更新了数据的。不管它是 INSERT 语句，还是 REPLACE 语句，抑或是 UPDATE 语句，对于增量更新语句的生成，当然也可以直接使用 UPDATE...SET...这样的语句即可保证数据的一致性。

表名交换

一个在线改表过程，上面已经讲完了关键的前两步，分别是搬迁数据及增量应用。那么，剩下的最后一部分就是交换表名了，这是最关键的。

pt-online-schema-change

pt-online-schema-change 工具在实现这部分的时候，是很容易处理的。因为增量数据是通过触发器来做的，也就是说，在线业务的修改操作和增量应用的处理是在同一个事务中完成的，只要搬完了数据，就可以直接去做 RENAME 操作了。因为 RENAME 操作会锁表，在锁表之间的写入，都通过触发器同步到新表了。而在锁表之后的写入，则会通过事务排队，被 RENAME 操作阻塞，等待 RENAME 操作完成之后，才可以继续写入。而此时，因为 RENAME 操作，触发器的触发表，也被修改为 _old 表的触发器了，所以不会影响在 RENAME 操作之后的请求。就这样，pt-online-schema-change 很简单地实现了表名交换操作，成功完成了 ALTER TABLE 操作。

Inception 内置 OSC

对于通过 Binlog 来处理增量的方法，在交换表名的方式上，与 pt-online-schema-change 的处理完全不同。一个最主要的原因就是增量的应用与在线业务写入不是在同一个事务中，因

为增量是通过分析 Binlog 拼接 SQL 而实现的事后型处理，所以没办法做成一个事务。

基于这样的原因而产生的一个问题就是，在收尾处必须要通过显式的锁表操作来实现让在线业务写入的阻塞。在处理完所有的 Binlog 之后，才能去做 RENAME 操作，才能保证不丢数据，以及改前改后的数据完整性和一致性。而这样做的缺点很明显，就是需要锁表，这个锁表时间的长短可以通过参数来设置。当然，这也与当前表的写入压力有关系，主要是看 Inception 如何处理，目前有多个参数可以控制这方面的安全性。

上面的问题说起来很简单，如果能真像上面所说的处理，也就不会引发本节所说的问题了。问题的起源，也就在这里。

先看一个截图，如图 43.1 所示。

```
mysql> lock tables test write, _FDC67936_test_new write;
Query OK, 0 rows affected (0.03 sec)

mysql> rename table test to _FDC67936_test_old, _FDC67936_test_new to test;
ERROR 1192 (HY000): Can't execute the given command because you have active locked tables
or an active transaction
mysql>
```

图 43.1

这里，我只能说：“想象很丰满，现实很骨感！”

设计了完美的解决方案，MySQL 却不允许这么做，怎么办？

这时，可以想想目的是什么。主要目的是要在 Lock 表之后，阻塞新在线业务的数据写入。有一个时间窗口，能够处理完已经产生的被改表的 Binlog，处理完之后，再去做 RENAME 表的操作。但 MySQL 不允许在一个事务中做这件事，所以为了实现改表的目的，必须要找另外的方式。要求 MySQL 解释并且允许这样的操作是不太现实的。关于这个问题就要另说了。

为了达到目的，想了一个不太完美的方法。如果把上面的一系列操作，分为两部分，将一部分交给当前处理增量的连接，只负责锁表，以及在锁表之后处理剩下的增量写入，将另一部分（也就是 RENAME 表的操作）交给其他连接，这样的方案是否可行？好像是可以的，下面根据图 43.2 及下面的文字解释来看一下详细步骤。

- 第 1 行表示 Master Thread 的 Lock 表操作开始，上锁对象是 origin 表、new_table 表。
- 第 2 行表示 Master Thread 处理增量已经结束。
- 第 3 行表示 Master Thread 在完成增量处理之后，通知 Rename Thread 去做 RENAME 操作。而如果执行了的话，RENAME 操作肯定会被阻塞，因为 Master Thread 已经给表上了锁。RENAME 语句为 RENAME TABLE origin to old_table, new_table to origin。

```
[Master thread] Lock origin table, new table and magic table, retry: 0
[Master thread] The end binlog position is (mysql-bin.000233:472580146)
[Master thread] Notify the rename thread to start work
[Master thread] Try to catch connection_id
[Rename thread] Rename table thread is awake
[Rename thread] Rename table thread get connection_id: 13156
[Rename thread] Rename table start to blocking, timeout: 16
[Master thread] Get the rename connection_id: 13156
[Master thread] Found the blocked rename table thread
[Master thread] Drop magic/old table
[Master thread] Unlock tables
[Rename thread] Successfully altered `sbtest`.`sbtest1`.
```

图 43.2

- 第 4 行表示 Master Thread 线程开始在 Processlist 中捕获 Rename 语句了, 等待并获取 connection_id (thread_id)。
- 第 5 行表示 Rename Thread 线程收到了 Master Thread 的提醒, 开始去做 Rename 操作了。
- 第 6 行表示 Rename Thread 线程拿到自己的 Rename 操作线程号, 并且告诉 Master Thread。
- 第 7 行表示 Rename 操作成功被阻塞了。
- 第 8 行表示 Master Thread 已经成功收到了 Rename 语句执行的 connection_id 号。
- 第 9 行表示 Master Thread 通过语句: `SELECT COUNT(*) FROM information_schema.PROCESSLIST WHERE ID=connection_id AND COMMAND = 'Query' AND STATE LIKE 'Waiting for table metadata lock%%'` 成功得到了被阻塞的 Rename 操作。
- 第 10 行表示, 如果上面成功找到了被阻塞的 Rename 语句, 则 Master 线程就可以删表了。先记下这一行, 后面会讲到。
- 第 11 行表示, 上面都做完了就放锁, 放锁之后, Rename 线程马上就返回了。因为放锁之后 Rename 操作就可以正常执行了。
- 表示改表已经完成, 由 Rename Thread 来决定。

到这里, 改表已经完成, 虽然过程有点坎坷, 但还是成功了。

做到这一点之后, 发现一个问题。假如在 Master 线程做完了通知 Rename Thread 之后, Rename Thread 还没有开始做 RENAME 操作, 这时如果 Lock Table 的 Master Thread 连接断了, 则此时锁表阻塞就会失效, 并产生新的增量数据。正在这时, RENAME 语句被执行了, 那么这部分增量就不会被同步到新表中去, 这部分数据就算丢了。为了防止并解决这个问题, 需要在改表一开始就把 old 表建立起来。这样的话, 即使出现这样的问题, 也会因为在 RENAME 执行的时候, 由于改名时目标表名已经存在而导致修改失败, 从而避免数据丢失。

在 Lock 表的时候（也就是在图 43.2 的第 1 行），也会将 old 表上锁。同样地，在图 43.2 中的第 10 行，看到有一步要删 old 表的过程，这正是在所有事情完成之后，将 old 表删除的操作。这里，只要将这个表删除，即使 Master Thread 对应的连接断了，也没有问题，因为 RENAME 操作已经被成功阻塞了。然后再去 UNLOCK TABLES。

到这里，改表的事情貌似在坎坷中一路走过来了，并且成功解决了所有问题。下面看一个完整的日志内容，如图 43.3 所示。

```
alter table sbtest1 engine innodb
  SQLSHA1: *E468035A553CC9F5A5A25189B153143791556402
  Percent: 100
  Remain_Time: 0(s)
  Information: [Master thread] Create new table: sbtest_786C6DA2_sbtest1_new completely
[Master thread] Create magic/old table: sbtest_786C6DA2_sbtest1_old completely
[Master thread] Get the binary log position before copy rows: mysql-bin.000232:54340775
[Master thread] No slaves found
[Master thread] Create copy rows thread completely
[Master thread] Create binary logs catch thread completely
[Master thread] Create the rename table thread
[Copy thread] Start to Copy rows, batch size: 200
[Copy thread] Start to get connection, retry: 0/3
[Master thread] Start to consume the increment SQL(s)
[Binlog thread] Binlog catch start with position: (mysql-bin.000232:54340775)
[Binlog thread] Start to read the binlog events and produce the increment SQL(s)
[Binlog thread] Get the connection and Dump binlog, retry: 0/3
[Rename thread] Rename table thread start to waiting
[Copy thread] Copy rows completely, copy rows: 5000000
[Copy thread] Copy thread exit
[Master thread] Lock origin table, new table and magic table, retry: 0
[Master thread] The end binlog position is (mysql-bin.000233:472580146)
[Master thread] Notify the rename thread to start work
[Master thread] Try to catch connection_id
[Rename thread] Rename table thread is awake
[Rename thread] Rename table thread get connection_id: 13156
[Rename thread] Rename table start to blocking, timeout: 16
[Master thread] Get the rename connection_id: 13156
[Master thread] Found the blocked rename table thread
[Master thread] Drop magic/old table
[Master thread] Unlock tables
[Rename thread] Successfully altered `sbtest`.`sbtest1`
[Master thread] Increment SQL(s) consume over, exactly 0 rows, including 0 insert(s), 0 update(s), 0 delete(s)
[Rename thread] Locked table for 118(ms)
[Rename thread] Rename thread exit
[Binlog thread] Binlog catch thread exit

Execute_Time: 62.790
```

图 43.3

内容是通过 Inception 命令 `inceptoin get osc processlist;` 获得的。

问题如果就这样解决了，那真是皆大欢喜了。但以上所述仅限于在 MySQL 主从及单机版本下的操作是没有问题的，而对于 Galera Cluster 类型的架构，问题就大了。

Galera Cluster 中的问题

Galera Cluster 在前面几个章节中已经详细讲过了，现在应该已经为我们所熟知了。不管是其优点、缺点，还是使用方式及如何避免一些所谓的坑，都已经是非常清楚的了。

而今天在线改表所引发的问题，其实还是与它的缺点有关系，那就是 DDL。

上面的 Inception 程序，在 Galera Cluster 上随便改一个表，就会发现改的过程不是很顺利。可能会发现，即使改一个空表也会等很久。但是，这里的问题是等很久也等不到。其实意思就是说根本不能返回来，如果有耐心的话，可以一直等下去。

怎么会在 Galera Cluster 中被卡死呢？首先看一下 Processlist，如图 43.4 所示。

```
mysql> show processlist;
302 | zhufeng.wang | 127.0.0.1:35096 | test1 | Query | 48 | Preparing for TO isolation |
drop table _B1893594_sbtest1_old | 0 |
0 |
305 | zhufeng.wang | 127.0.0.1:35180 | test1 | Query | 62 | Waiting for table metadata lock |
rename table sbtest1 to _B1893594_sbtest1_old, _B1893594_sbtest1_new to sbtest1 | 0 |
```

图 43.4

很明显可以看到，drop table 的语句状态为 Preparing for TO isolation。如果理解了之前讲的 Galera 内容的话，这里应该就会很清楚是什么意思了。更详细的信息可以看一下这个线程的堆栈，如图 43.5 所示。

```
Thread 6 (Thread 0x7f554485c700 (LWP 27565)):
#0 0x0000003999e0b63c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
#1 0x00007f5bb3bb94c in galera::Monitor<galera::ReplicatorSMM::CommitOrder>::enter(galera::ReplicatorSMM::CommitOrder&) () from /home/q/mysql/galera/lib/libgalera_smm.so
#2 0x00007f5bb3bb1ca8 in galera::ReplicatorSMM::to_isolation_begin(galera::TrxHandle*, wsrep_trx_meta*) () from /home/q/mysql/galera/lib/libgalera_smm.so
#3 0x99907f5bb3bc509c in galera_to_execute_start () from /home/q/mysql/galera/lib/libgalera_smm.so
#4 0x00000000005b83c7 in wsrep_to_isolation_begin(THD*, char*, char*, TABLE_LIST const*) ()
#5 0x000000000070e33b in mysql_execute_command(THD*) ()
#6 0x0000000000710388 in mysql_parse(THD*, char*, unsigned int, Parser_state*) ()
#7 0x0000000000710512 in wsrep_mysql_parse(THD*, char*, unsigned int, Parser_state*) ()
#8 0x0000000000712943 in dispatch_command(enum_server_command, THD*, char*, unsigned int) ()
#9 0x0000000000713e1b in do_command(THD*) ()
#10 0x000000000006daf6f in do_handle_one_connection(THD*) ()
#11 0x000000000006db157 in handle_one_connection ()
#12 0x000000000009a1a4a in pfs_spawn_thread ()
#13 0x0000003999e07a51 in start_thread () from /lib64/libpthread.so.0
#14 0x0000003999ae896d in clone () from /lib64/libc.so.6
```

图 43.5

可以看到，执行当前事务的线程是想要进入 Commit 临界区，但是前面已经有一个事务了，就是 RENAME 语句。而它的 GTID 大于 RENAME 的 GTID，因为 Commit 是顺序的，所以 DROP TABLE 就要等待了。

而此时来看一下 RENAME 操作的堆栈，如图 43.6 所示。

```
Thread 3 (Thread 0x7f5544799700 (LWP 28803)):
#0 0x0000003999e0b63c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
#1 0x00000000006776db in MDL_wait::timed_wait(MDL_context_owner*, timespec*, bool, PSI_stage_info_v1 const*) ()
#2 0x000000000067a247 in MDL_context::acquire_lock(MDL_request*, unsigned long) ()
#3 0x000000000067a60b in MDL_context::acquire_locks(I_P_List<MDL_request, I_P_List_adapter<MDL_request, &(MDL_request::next_in_list), &(MDL_request::prev_in_list)>, I_P_List_counter, I_P_List_no_push_back<MDL_request> >*, unsigned long) ()
#4 0x00000000006bce8e in lock_table_names(THD*, TABLE_LIST*, TABLE_LIST*, unsigned long, unsigned int) ()
#5 0x0000000000728f46 in mysql_rename_tables(THD*, TABLE_LIST*, bool) ()
#6 0x000000000070ce32 in mysql_execute_command(THD*) ()
#7 0x0000000000710388 in mysql_parse(THD*, char*, unsigned int, Parser_state*) ()
#8 0x0000000000710512 in wsrep_mysql_parse(THD*, char*, unsigned int, Parser_state*) ()
#9 0x0000000000712943 in dispatch_command(enum_server_command, THD*, char*, unsigned int) ()
#10 0x0000000000713e1b in do_command(THD*) ()
#11 0x00000000006daf6f in do_handle_one_connection(THD*) ()
#12 0x00000000006db157 in handle_one_connection ()
#13 0x00000000009a1a4a in pfs_spawn_thread ()
#14 0x0000003999e07a51 in start_thread () from /lib64/libpthread.so.0
#15 0x0000003999ae896d in clone () from /lib64/libc.so.6
```

图 43.6

很明显可以看到，执行当前事务的线程是在等待表锁。而这个表锁，是在 Master Thread 显式上的。已经在等锁了，说明 RENAME 操作已经在 Galera 的 Commit 临界区中了，也就是说 RENAME 产生的 GTID 小于上面 DROP TABLE 的 GTID，这样才能产生等待。为了证明这一点，来看一下 RENAME 操作的代码，如下。

```
case SQLCOM_RENAME_TABLE:
{
    TABLE_LIST *table;
    for (table= first_table; table; table= table->next_local->next_local)
    {
        if (check_access(thd, ALTER_ACL | DROP_ACL, table->db,
                        &table->grant.privilege,
                        &table->grant.m_internal,
                        0, 0) ||
            check_access(thd, INSERT_ACL | CREATE_ACL, table->next_local->db,
                        &table->next_local->grant.privilege,
                        &table->next_local->grant.m_internal,
                        0, 0))
            goto error;
    }
}
```

```

/* 重点关注下面这句,说明在RENAME执行前,首先会进入Commit临界区 */
WSREP_TO_ISOLATION_BEGIN(0, 0, first_table)
if (mysql_rename_tables(thd, first_table, 0))
{
    goto error;
}
break;
}

```

通过上面的代码,可以看到 RENAME 在等待锁的时候确实已经进入了临界区,而 DROP TABLE 没有进入,说明它的 GTID 必然是更大的。

到这里,已经明确地知道,当前的 Galera Cluster 节点已经死锁且出现了互相等待的状态。在第 29 章中已经讲过,对于 DDL 的执行,只要开始执行了,那么在 Processlist 中不管是使用 kill connection thread_id 命令,还是使用 Control+C 的方式,都无济于事。所以,已经拿这个节点没办法了,除非杀了它。

一个有趣的实验

这里,再做一个有趣的实验。已经清楚了发生死锁的过程,那么现在就模拟一个场景,手动让这个问题重现。

在 Galera Cluster 上开一个会话(叫会话 1),先执行一条 Lock 表的语句,如下。

```

mysql> lock tables sbtest1 write, _B1893594_sbtest1_old write,
        _B1893594_sbtest1_new write;
Query OK, 0 rows affected (0.00 sec)

```

然后再开一个会话(会话 2),执行一条 RENAME 语句,如下。

```

mysql> rename table sbtest1 to _B1893594_sbtest1_old, _B1893594_sbtest1_new to
        sbtest1;
——悄悄地说了声:我被阻塞啦!

```

此时在会话 1 中再执行 DROP TABLE _B1893594_sbtest1_old 语句,如下。

```

mysql> drop table _B1893594_sbtest1_old;
——感觉不太对,也执行不下去了。

```

其实,此时死锁已经发生,具体如下图所示。

会话 1 如图 43.7 所示。

```
mysql> lock tables sbtest1 write, _B1893594_sbtest1_old write, _B1893594_sbtest1_new write;
Query OK, 0 rows affected (0.00 sec)

mysql> drop table _B1893594_sbtest1_old;
```

图 43.7

会话 2 如图 43.8 所示。

```
mysql> rename table sbtest1 to _B1893594_sbtest1_old, _B1893594_sbtest1_new to sbtest1;
```

图 43.8

此时，Processlist 中的内容和之前所述的是完全一样的，如图 43.9 所示。

```
mysql> show processlist;
| 302 | zhufeng.wang | 127.0.0.1:35096 | test1 | Query | 48 | Preparing for TO isolation | 0 |
| 305 | zhufeng.wang | 127.0.0.1:35180 | test1 | Query | 62 | Waiting for table metadata lock | 0 |
```

图 43.9

或许有人想到了，可以先使用 Control+C 中止当前语句的执行，比如 DROP TABLE 语句，然后在命令行中执行 UNLOCK TABLE 语句。想法是不错，但请看图 43.10。

```
mysql> lock tables sbtest1 write, _B1893594_sbtest1_old write, _B1893594_sbtest1_new write;
Query OK, 0 rows affected (0.00 sec)

mysql> drop table _B1893594_sbtest1_old;
^C^C^C -- sending "KILL QUERY 302" to server ...
Ctrl-C -- query aborted.
^C^C^C -- sending "KILL 302" to server ...
Ctrl-C -- query aborted.
ERROR 2013 (HY000): Lost connection to MySQL server during query
mysql> unlock tables;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id: 332
Current database: test1

Query OK, 0 rows affected (0.00 sec)
```

图 43.10

之后，执行是成功了，但是看了看 processlist 还是一样。这是为什么呢？其实很简单，会话 1 已经通过 Control+C 被取消了，但由于这个线程已经被阻塞在了 Galera 层，不能返回来了

(会话 1 没有被真正杀死)，所以能够成功执行 UNLOCK TABLE 是通过 MySQL 客户端新开了一个会话（会话 3）做到的。但是对会话 1 没有任何影响，继续死锁。

到现在为止，已经成功地在 Galera Cluster 上构造了一个死锁的问题，那么这个实验这样就够了么？并不够，继续！

假如现在在另外两个节点中的一个节点上，一直对一个数据量比较大的表执行删除操作，每次只删除 1 条，为的是用最小的数据量构造最多的事务数，如图 43.11 所示。

```
mysql> show table status like "%_A52D502B_sbtest1_new%" \G
***** 1. row *****
      Name: _A52D502B_sbtest1_new
      Engine: InnoDB
      Version: 10
      Row_format: Compact
      Rows: 86040446
      Avg_row_length: 233
      Data_length: 20066598912
      Max_data_length: 0
      Index_length: 2164211712
      Data_free: 5242880
      Auto_increment: 261564307
      Create_time: 2016-10-26 17:11:35
      Update_time: NULL
      Check_time: NULL
      Collation: utf8mb4_general_ci
      Checksum: NULL
      Create_options: max_rows=1000000
      Comment:
1 row in set (0.00 sec)

mysql> delete from _A52D502B_sbtest1_new limit 1;
Query OK, 1 row affected (0.00 sec)

mysql> delete from _A52D502B_sbtest1_new limit 1;
Query OK, 1 row affected (0.00 sec)
```

图 43.11

删除语句一直做，能做多少次做多少次。做的过程中，可以观察一下发生死锁的节点的集群状态，如图 43.12 所示。

```
mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 76 |
+-----+
1 row in set (0.00 sec)

mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 94 |
+-----+
1 row in set (0.00 sec)

mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 130 |
+-----+
1 row in set (0.00 sec)

mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 400 |
+-----+
1 row in set (0.00 sec)

mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 922 |
+-----+
1 row in set (0.00 sec)

mysql> show status like "wsrep_local_rcv_queue";
+-----+
| Variable-name | Value |
+-----+
| wsrep_local_rcv_queue | 1025 |
+-----+
1 row in set (0.00 sec)
```

图 43.12

可以看到，在不断插入数据的过程中，这个值一直在变，直到增加到 1025 之后，就不再变了。而此时在做 delete 操作的节点发生了什么呢？如图 43.13 所示。

```
mysql> delete from _A52D502B_sbtest1_new limit 1;
Query OK, 1 row affected (0.01 sec)

mysql> delete from _A52D502B_sbtest1_new limit 1;
Query OK, 1 row affected (0.00 sec)

mysql> delete from _A52D502B_sbtest1_new limit 1;
Query OK, 1 row affected (0.01 sec)

mysql> delete from _A52D502B_sbtest1_new limit 1;
```

图 43.13

哦？被卡住了？再来看集群状态，有没有异常的参数等，如图 43.14 所示。

```
mysql> show status like "wsrep_flow_control_paused%";
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_flow_control_paused_ns | 263569846174 |
| wsrep_flow_control_paused | 0.004753 |
+-----+-----+
2 row in set (0.00 sec)

mysql> show status like "wsrep_flow_control_paused%";
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_flow_control_paused_ns | 263896729973 |
| wsrep_flow_control_paused | 0.004759 |
+-----+-----+
2 row in set (0.00 sec)

mysql> show status like "wsrep_flow_control_paused%";
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_flow_control_paused_ns | 264232853693 |
| wsrep_flow_control_paused | 0.004765 |
+-----+-----+
2 row in set (0.00 sec)
```

图 43.14

可以看到这个集群中每一个节点的两个值在一直增大。这两个参数表示的是节点是否发生了 Flow Control，一直增大则说明 Flow Control 已经发生了，但一直没有恢复。这就麻烦了，

整个集群不能再写入任何事务，因为只要写入，就会被卡住，也就是说，此时整个集群都死锁了。

问题出现了，就总要解决！那么，怎么解决呢？

解决方案

说白了，这个问题是因为集群中某个节点的死锁，导致了整个集群的 Flow Control，进而使整个集群发生了死锁，并且永远没办法自己解开。

那么，如果碰巧在线上做了类似的操作导致了这样的问题，该如何解决呢？

之前详细介绍过 Flow Control 的产生、避免方法及处理方法等，现在就需要通过这些处理方法来处理。首先，要找出是谁引起了 Flow Control。之前已经知道，Flow Control 一般都是因为某一个节点执行太慢，把其他节点拖累了，这个问题也符合这个特点，已经慢到了极点。那么在这个问题中，发生死锁的节点很明显是 Flow Control 发出者，而这个节点已经没有办法自动恢复，或者是手动在线恢复了，只能像上面所说的，杀了它！

赶紧的，故障了，快点解决：kill -9 mysqld_pid …

咦？故障消除了。就是这么神奇！

但是一定要记住，在做 kill 之前，别忘记做最后一件事，那就是看看这个节点的状态值，如图 43.15 所示。

```
mysql> show status like "wsrep_last_committed";
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_last_committed | 1336741102 |
+-----+-----+
1 row in set (0.00 sec)
```

图 43.15

这一点很有用，不然会导致需要做 SST。

故障恢复了，那么这个节点还需要恢复服务。此时，将上面的 wsrep_last_committed 值填到 grastate.dat 文件的 seqno 中，并且把其他节点中找到的 wsrep_cluster_state_uuid 值填到 grastate.dat 文件的 uuid 的位置，这样相当于将节点被杀死之前的点找到了，然后就可以再次启动数据库了。很快就可以启动了，并恢复了之前的平静，好像一切事情都没有发生过。

总结

这里的关键问题在于删表操作。也就是为了防止丢数据，先将 old 表建立起来，在 UNLOCK TABLES 语句执行前再将其删除，保证 RENAME 可以正常进行。如果没有这个删表操作，则 RENAME 只是会被阻塞。在 UNLOCK 之后，它也会正常执行下去，从而不会造成这样的局面。

那么这个删表操作可以不执行么？只能说，不怕一万就怕万一。这个连接不经常断，但有时候断得就是那么巧。有可能会断，只能说这是小概率事件。

那么如果是这样的话，是不是表示这种改表方式在 Galera Cluster 上就被判死刑了呢？其实可以这么说，也可以说是死缓，因为还有一种方式可以让这件事情从一开始就不会走这么复杂的路。

那就是让图 43.16 中的操作不要报错，可以正常正确地执行下去。

```
mysql> lock tables test write, _FDC67936_test_new write;
Query OK, 0 rows affected (0.03 sec)


mysql> rename table test to _FDC67936_test_old, _FDC67936_test_new to test;
ERROR 1192 (HY000): Can't execute the given command because you have active locked tables or an active transaction
mysql>
```

图 43.16

此时一切都明白了。这样的话，压根就不会涉及多个会话的事了，所有事情都自己完成就没有这样的问题了。

可是，MySQL 当初这样设计的想法是什么呢？也许深究一下代码可以发现其中原委。但是这有什么用呢？如果 MySQL 一直这样拦着，作为一个外围工具只能无话可说。

相关代码如下。

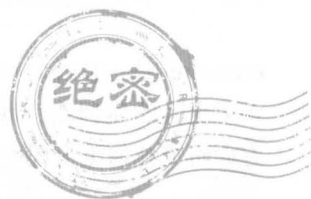
```
 bool mysql_rename_tables(THD *thd, TABLE_LIST *table_list, bool silent)
{
    /*
     * Avoid problems with a rename on a table that we have locked or
     * if the user is trying to do this in a transaction context
     */
    if (thd->locked_tables_mode || thd->in_active_multi_stmt_transaction())
    {
        my_message(ER_LOCK_OR_ACTIVE_TRANSACTION,
                   ER(ER_LOCK_OR_ACTIVE_TRANSACTION), MYF(0));
```

```
    DEBUG_RETURN(1);  
}  
/* other code lines ... */  
}/*mysql_rename_tables*/
```

这下了然了，看代码可以解决一切问题。从上面代码注释中可以看到，为了防止出现问题，在一个有加锁操作的事务中，不能去做 Rename Table 操作。当然，不管改几个表，只要前面有上锁操作，就都是不行的。

其实可以想一下，既然已经给当前所需要的表加锁了，为什么不能去做 RENAME 操作呢？是不是因为 DDL 是自动提交的，所以提交之后会导致锁失效呢？那么，如果这样的结果被每个人都熟知并且能接受，是不是也就没有问题了？种种如此猜想，如果能变成现实，那么在在线改表的问题上，可能就会更进一步。

当然，这个问题的发生，也不只是 MySQL 的问题，Galera 也存在问题。对于 DDL 的执行，首先是强制进入了 TOI (Total Order Isolation)，导致所有相关其他节点的事务都会被杀死，并且整个改表操作都会在 Commit 的临界区做，导致其他事务的提交都做不了。这也是 Galera 一直被诟病的问题。不过，听说 Galera 也一直致力于解决这两个问题。那么在此，也希望它在传说中的 Galera 4.0 版中能有很好的表现，有可能这个问题也就顺其自然地被解决了。



第三部分 Inception 篇



内参〔2017〕3号

Inception 是一款针对 MySQL 的 SQL 语句审核自动化运维工具。使用 Inception, 将会给 DBA 带来更大的便利性, 将 DBA 从繁冗的工作中解放出来, 做更多的自动化工作, 或者从架构方面研究如何更大程度地保证数据库的高可用等。

MySQL 语句的审核在业界已经基本被认同, 这实际上也是对 MySQL 语句写法的统一化、标准化。而依照这个标准做人工审核其实是很吃力的, 标准越多, DBA 越累, 开发也越累。

在这个追求自动化运维的时代, 审核也必须要跟进步伐, 因此 Inception 诞生了。

Inception 远不止是一个自动化审核工具, 同时还具备执行、生成影响数据的回滚语句(类似闪回的功能)这样一条龙服务的功能。这给 DBA 的工作带来了翻天覆地的变化, DBA 从此就从繁重的审核、连接 MySQL Server 执行、出错后很难回滚(如果提前没有备份的话)的被动局面中解放了出来。有了 Inception 之后突然发现, 做 DBA 原来可以这么轻松, 工作可以不再总是重复劳动, 节省了大量的时间, 也就有更多的自由时间去学习, 进一步向自动化运维平台的实现等更智能化的方向去发展, 是具有里程碑意义的。

2015 年 8 月, 我们本着回报开源社区的初心, 开源了 Inception 项目。对于开源, 我们是有敬畏之心的。就像当时的领导吴永强先生所说:“要么你就不开源, 安安静静地自己用。如果开源的话, 就一定要把这件事情做好——项目怎么维护, 代码怎么合并, 受众怎么培训, 问题如何解决等。”为此, 我们做了如下一些工作。

- 大规模重构 Inception 代码, 剥离公司相关的业务和敏感信息。
- 核心功能模块化和接口化, 特别是对最重要的审核规则模块。我们把它彻底剥离出来做成接口, 以便贡献者提交功能性代码。
- 建立开源项目的咨询平台。目前我们的社区有数百人之多, 使用 Inception 的用户遍布了国内大部分互联网公司。
- 认真编写 Inception 文档, 争取让所有的 Inception 用户都能在文档的帮助下轻松自如地使用 Inception。

经过将近两年的不断积累和总结, 我们深知 Inception 的价值和对使用它的 DBA 的帮助。因此, 我们决定把使用与开发 Inception 的经验集结, 作为本书的重要组成部分, 以期能够帮助更多的读者。

Inception 开源 GitHub 地址: <https://github.com/mysql-inception/inception.git>。

44

Inception 诞生记

关于 SQL 审核

MySQL 语句需要审核，这一点每个 DBA 及开发人员都懂。但鉴于 SQL 语句及环境的复杂性，审核的方法多种多样，可以说每个公司可能都有自己的方法。下面介绍两种之前常用的审核方法。

半自动化方法

在 Inception 出现之前，很多公司也已经进入了审核自动化阶段。但这种自动化实际上还只能算是半自动化，因为在各个方面，例如其使用的友好性、审核的准确性及工作效率等，都还比较难让人满意。不过，这已经有了很大的进步，正所谓：It's better than nothing。

那么半自动化方法是如何工作的，它的实现原理是什么呢？请先看图 44.1。

这种半自动化方法，一般包括四部分，分别如下。

- 一个很强大的审核程序。
- 线上待执行数据库服务器。
- 与线上数据库对应的 Beta 数据库服务器（定时与线上同步数据）。
- 执行前用来备份数据的备份服务器。

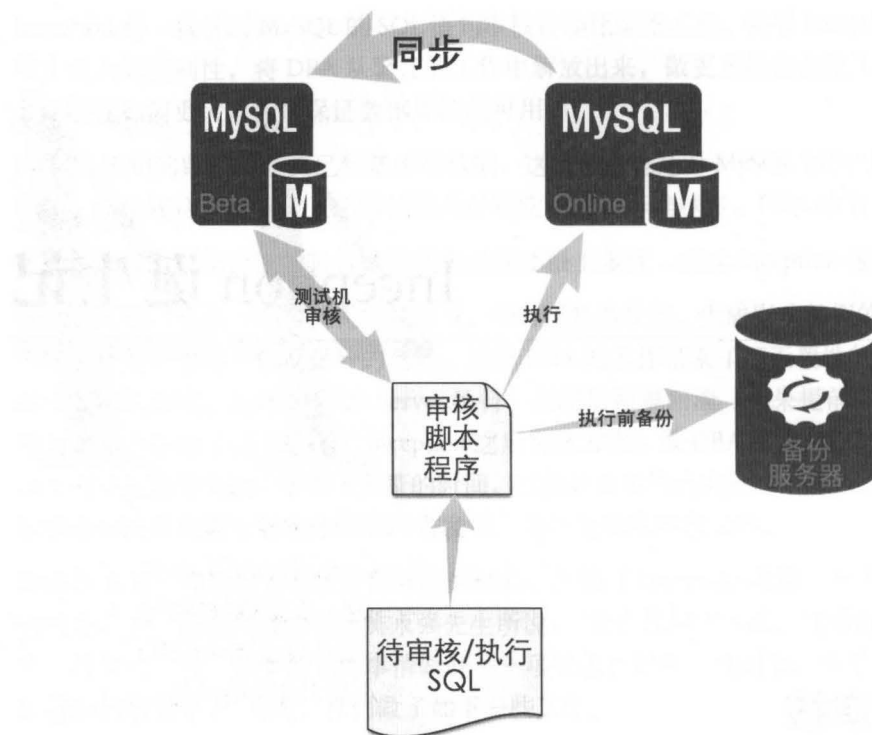


图 44.1

对于第一部分的审核程序，一般需要包括以下功能。

- 具有简单的语法、语义检查功能。但一般情况下，它都是做一些简单的匹配功能，而不是像 MySQL 一样，对 SQL 语句进行词法语法分析，做出精确的语法匹配及分析。所以，在这种情况下，很难做到精确审核。最简单的情况，就是提取出 SQL 语句涉及的数据库名、表名、列名及语句类型等，至于更深入复杂的分析，需要介入到 MySQL 的语法分析功能，这块是很复杂的，一般很难做到。
- 在提取到对应的库、表、列等对象信息之后，具备通过线上或 Beta 环境对信息进行比对的简单功能。这样可以提前判断一些很明显的错误，这方面要尽可能地多做工作，以提高审核效率。
- 具备提前分析过滤出一些高危 SQL 语句的功能，提高审核效率。比如通过提前在线上或 Beta 数据库上执行 EXPLAIN 命令等方式，一方面可以检验语法及语义的正确性，另一方面可以根据影响行数、索引使用等返回信息，提前过滤出可能对数据库影响比较大或语法语义错误的语句，从而提高审核效率。

SQL 审核功能非常多，就不一一列举了，正所谓：只有你想不到的，没有你做不到。

但是做再多，总是感觉走了弯路，总是力不从心，总是在不断地补洞，总是有补不完的洞，或者有些洞根本就补不上。

实际上，这种方式从一开始就有问题，会造成程序的可维护性差、效率低下、准确性不高等一系列问题。

对于第二部分的线上数据库服务器，由于第一部分的审核程序有可能存在没有审核到的漏洞，导致一些本身存在问题的语句被认为没有问题而审核通过，这无疑给线上数据库带来了很大的风险，所以审核的准确性一定不能忽视。

对于第三部分，因为审核程序首先要在 Beta 环境中执行一次，发现没有问题后才会去线上执行。那么这个 Beta 就需要时刻与线上环境保持同步，数据需要一样，数据库及表的元数据也必须要保证相同，这样在 Beta 中执行才具有参照性，不然有可能在 Beta 上不存在问题，而在线上执行时出了问题。那么，这里的关键问题是如何能保持时刻同步的问题，比如在一个修改比较频繁的数据库实例上，在 Beta 中执行过的操作发现问题，最终没有在线上执行，那么当这个实例再有新的审核时，无效操作如何回滚，是否能够保证不影响后面的执行呢？在数据量很大的情况下，不管是执行还是回滚，都是问题。在数据被频繁使用过之后，Beta 环境有可能需要不定时地重新做镜像，那么是不是还需要在晚上低峰的时候做这个镜像？在高峰的时候是不是会影响线上？如此种种，实际上都是很难解决的问题，如果走了这条路，恐怕这将是 DBA 的又一个噩梦。

对于最后一部分，为了防止上面第三部分执行之后（不管是线上库还是 Beta 数据库），导致数据不能回滚或恢复的问题，最好是在执行前将要影响的数据做个备份，这样万一出了问题，还可以及时回滚以尽量减少对业务的影响时间。但是，这个备份说起来简单，真正去做的时候，却发现真的是没办法下手。在这里可能有下面三种方法。

- 通过审核程序想办法先查出 SQL 语句将要影响的数据，生成反向的回滚语句。
- 备份整个表，修改失败或需要回滚的时候，直接使用这个备份表即可。
- 利用 Binlog，通过类似闪回的功能实现数据回滚。

但在上面的方法中，都有不同程度的复杂性及不可行性。

- 先说第一种方法。这种方法需要审核程序做更多的工作，在执行前取出 SQL 语句将要影响的全量数据，导出来备份，然后再去修改。但这种方法存在很多的问题：首先，不是所有的增删改语句都能简单地转换为一个查询语句，然后通过这个查询语句将数据导出来；其次，就是准确性问题，因为数据一直在变，数据量大的话，花长时间导出之后，再去执行，实际上执行影响的数据已经与备份的数据不相同了，所以备份已经没有太大意义了；最后，就是审核程序自己不知道当前 SQL 语句具体修改了哪些数据，或者哪些列，当数据需要恢复时，它不知道该如何恢复。

- 再说第二种方法。很明显,这种方法对于数据量比较小的表是可行的,因为它是某一个时刻的完整备份的副本。但如果数据量大一点,备份这个表的工作量就非常大了。
- 而再来看第三种方法。很明显,这种方法是最好的,并且是最准确的,它有一个前提是审核程序去执行的时候生成的 Binlog 是 Row 模式的。不过,这种方法实现起来门槛非常高,需要全面兼容 MySQL Binlog 的格式,从文件中分析出数据的修改内容,同时还要在审核程序执行时,取得其执行时的 thread_id 及 Binlog 位置等信息,最后根据内容生成回滚语句。听上去非常完美,但很少有人会为了一个备份去做这么复杂的工作,所以这种方式在半自动化审核中一直没有被用起来。

因而,在半自动化审核中,备份基本上是一个空白。因为无论哪种方法,都很难实现得完美、可用。

综上所述,可以看出,半自动化方法实际上存在很多难解的问题,不易推广,每个公司都各自为战,所以导致业界没有一个可以公用的、统一的且被大家认可的审核工具。

人肉法

在 MySQL 的审核方法中,有一种方法是最传统、门槛最低,同时也是与半自动化方法共存的一种方法,那就是人肉法。所谓“人肉法”,就是所有的审核工作都由人工来完成。先来看一下一般的人工审核工作方式图,如图 44.2 所示。

这种方法的交流方式一般是邮件。需要执行变更时,DEV/QA 写邮件向 DBA 发起变更,在 DBA 收到之后,用“火眼金睛”目测他们写的 SQL 是不是有问题,这里的问题要包括如下几个方面。

- 语法错误:这些错误其实真是难为 DBA 了。即使 DBA 都是“火眼金睛”,也还是有可能看不出来一些潜在的错误,终究不是机器人。比如,经常有开发没有经过测试就发给 DBA,会丢失分号,或者将“”写为“”,这可真看不出来,只有真正到线上实践后,才会发现错误。
- 语义错误:这种错误需要 DBA 时刻到线上把表结构、数据量等拿出来,然后再与 SQL 语句中用到的对象比对,看看 SQL 语句中用到的列、表、库等是否正确。此时,也需要 DBA 高超的目测能力(也就是我们熟知的找茬能力),想想在一个列名中多/少一个字母或者空格,这能看得出来么?我想未必。
- 规则错误:对 MySQL 的 SQL 语句进行审核的目的就是想让 SQL 语句尽可能地统一,减少错误,优化 SQL 性能,提高 DBA 运维效率。实际上,DBA 定义的规则越多,DBA 越累。因为所有的 SQL 语句都要与规则进行比对,找到不符合规则的语句及问题,这有点类似一个成语叫“作茧自缚”。DBA 在每次拿到 SQL 语句后,心中都要遍历好几遍“规则宝典”。长时间的高 Load 运算已经导致 DBA 精疲力竭,而审核一个大项目又花了大

半天的时间，也就是说 DBA 的大部分时间都花在了审核上。这样效率太低，DBA 的产出也太低（但这些问题又不能不做）。

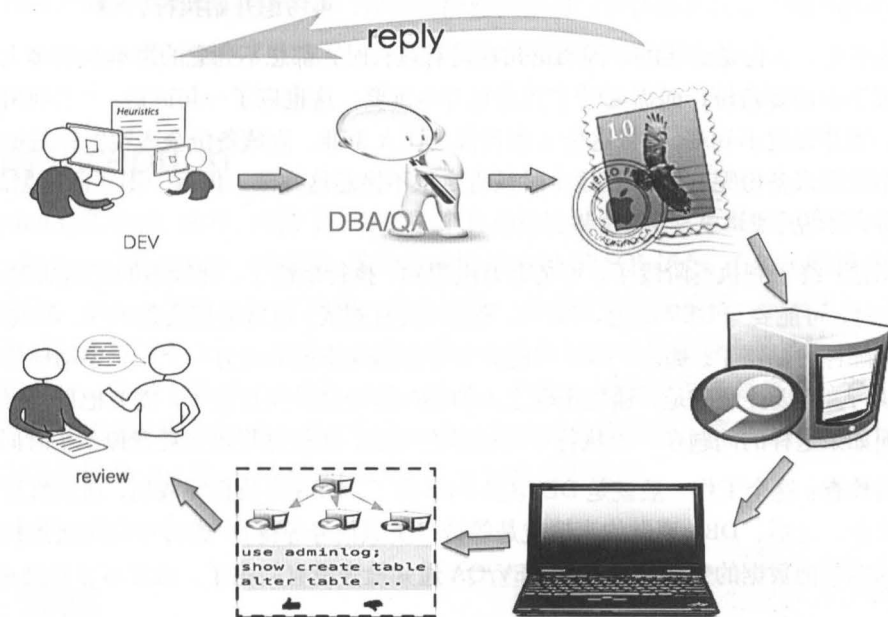


图 44.2

- 循环审核：DBA 在辛辛苦苦地找到力所能找到的错误之后，发邮件给 DEV（总算能休息一下了），同时希望这是“倒数第二次审核”（因为如果 DEV 再回邮件之后，一审核发现问题都已经改过了，并且没有新问题，该多好啊！所以谓之“倒数第二次审核”）。但是，经常是事与愿违，DEV 回复的邮件可能只会说一句“能改的都改好了”（哎，能告诉我哪些是不能改的么？），那么，DBA 就需要带着之前审核出来的结果，一个一个进行比对检查一下有没有改好（不管有没有改好都要看一遍，因为 DBA 也不知道他改了哪些），实际上和重新审核一遍没什么差别。因为这存在两方面的工作，一方面是审核之前提出来的问题有没有改好，第二方面是在改的过程中有没有引入新问题。从以上过程可以知道，DBA 在这方面的的工作量是非常大的，工作时间也基本都花在这上面，效率低下。
- 主观性：对于已经定义的规则，虽然明明白白写在那里，但不同的 DBA 对同一条规则的理解其实是不同的，所以存在宽松不均的问题。同一个开发，面对不同的 DBA 时，有时可以很轻松地通过，但有时又是铁律，搞得 DEV/QA 摸不着头脑，最终导致规则不成规则，很难推行下去。
- 执行前检查：经过以上的折腾后，好不容易通过了审核，可以执行了，DBA 现在需要做的事情就是先对线上环境做一次检查，比如 DEV 写的 IP 是读库的虚 IP 还是写库的

- 虚 IP，执行语句时对线上的影响多大，对于 DML 语句，如果影响行数太多，可能还需要拆开再去执行，而如果是 DDL，可能需要考虑晚上再去执行，或者用 OSC (pt-online-schema-change) 工具来执行等。这些问题确定之后，再考虑开始执行。
- 执行前备份：备份是必要的，因为语句在没有执行时，都想不到它的影响会有多大，一般情况下不需要备份，而需要时才知道它有多重要，这也应了一句谚语：“书到用时方恨少，事非经过不知难。”但这份工作也很是让人为难，应该备份全表呢？还是把影响的数据查出来备份呢？DBA 在这个时候肯定很不愿意这样做，但万一出了问题怎么办？都懂得，不说了。
 - 执行出错：终于到执行阶段了，可是万万没想到，执行出错了，比较多的是语法/语义错误。此时，可能要与 DEV 沟通，为什么发起时没有测试，通常他们会告诉你，测试过了，但贴到邮件时贴错了。那么，DBA 可能就需要把出现错误的地方一个一个手动地修改过来再执行。需要注意的是，错误出现之前的那些语句已经执行完了，需要把那些语句去掉，而如果这样的问题在一次执行中出现多次的话，执行过程也会耗费掉不少时间。
 - 执行后检查：这个工作一般就是 DEV/QA 的事情了。DBA 在执行完成后，通知执行完成并做检查。之后，DBA 要做的事情就是等待（类似信号等待），或者可以切换到其他线程去处理其他数据的变更请求。当 DEV/QA 通知检查没有问题了，这件事才算结束。

一个 DBA 一天的时间往往都被这些不想做而必须去做的事情困扰着，同时 DB 组还发现人力根本不够，需要招人（难道招人来了还是去做这些事情么？），DBA 难道就是只做这些事情的人么？当然不是！

不满现状的追求

本人在做过一段时间的 DBA 之后，发现 DBA 的工作根本苦不堪言，基本所有的时间都被这些繁冗的事情困扰着，根本没有多余的时间去做一些更高大上的工作，比如读一读源码等。

后来，听说一些大公司都有一些自动化审核工具，此时如拨云雾见青天，心想现在不正是自动化运维所兴起的时代么，这种工作完全可以由自动化工具所代替，想到这里不由得感到兴奋。

后来就开始调研他们审核工具的实现方式，但后来发现也基本都是上面提到的半自动化实现方法，感觉总是不怎么理想，DBA 的负担还是非常重。如何能优化？或许有一种更好的方案。

有一次，因为有一个大项目要上线，我正在审核一堆建表语句，已经心力交瘁了。突然，蹦出一个念头——MySQL 可以执行任何提交给它的 MySQL SQL 语句，并且存在任何问题都会提前报出来，不就是因为它支持这样的功能么（好像是废话）？问题的关键就在这里。既然 MySQL 可以执行那些 SQL 语句，那么是不是可以做一个和 MySQL 完全兼容的东西，去审核

这些 SQL 语句呢？或者说，如果把 MySQL 进行切割改造，让它不去执行，而是分析之后找出存在问题的地方，不就解决了审核不准确这个难题了么？想到这里，一股暖流流过心头，好像在远方的某个地方，看到了希望之火——哦，那是太阳！

现在，Inception 终于诞生了！它是一个关于 DBA 的美好的故事。

何谓 Inception

Inception 是集审核、执行、回滚于一体的一个自动化运维系统，它是根据 MySQL 代码修改过来的。用它可以很明确地、详细地、准确地审核 MySQL 的 SQL 语句，它的工作模式和 MySQL 完全相同，可以直接使用 MySQL 客户端来连接，而不需要验证权限。相对于应用程序（上层审核流程系统等）而言，它是一个服务器，在连接时需要指定服务器地址及 Inception 服务器的端口。而对于要审核或执行的语句所对应的线上 MySQL 服务器来说，它是一个客户端，在内部需要实时地连接数据库服务器来获取所需要的信息，或者直接在线上执行相应的语句及获取 Binlog 等。Inception 就是一个中间性质的服务。

图 44.3 所示为 Inception 的架构。

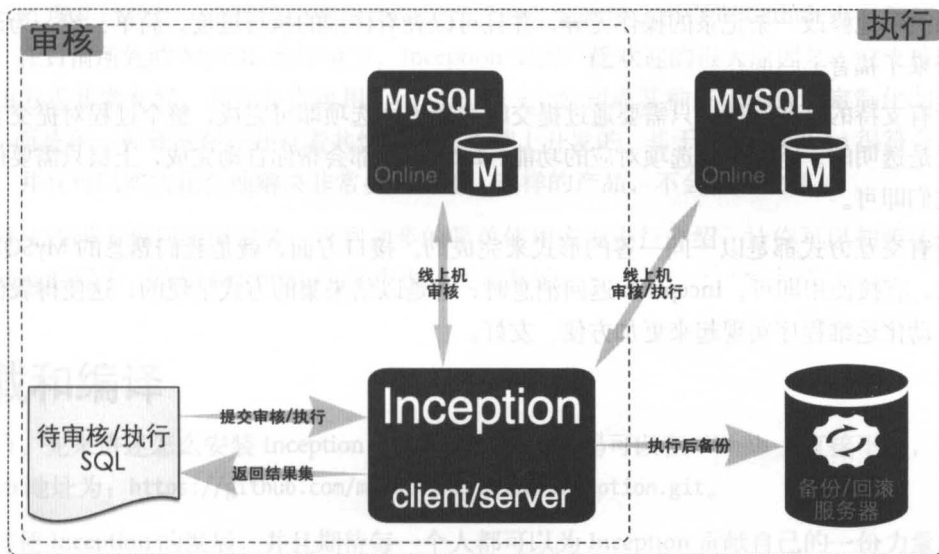


图 44.3

Inception 具备以下的丰富功能，这对于作为 DBA 的你来说，每一点都是足以让你看一眼就会心动的亮点。

- 可以通过对提交的所有语句做词法、语法分析，将存在问题的地方的错误信息返回给审核提交者。

- 当一个表、库、列等信息不正确或不符合规范，或者使用了一个不存在的对象等情况而报错时，提供语义分析。
- 提供了很多针对 SQL 规范性约束的内置功能。这些内置的规范约束，都可以通过系统参数来配置，可以根据自己的需求来设定。
- 提供 SQL 语句的执行功能，可执行的语句类型包括常用的 DML 语句、DDL 语句及 truncate table 等操作。
- Inception 在执行 DML 时还提供生成回滚语句的功能，对应的操作记录及回滚语句会被存储在备份机器上，备份机器通过配置 Inception 参数来指定。执行 DML 所生成的回滚语句是通过 Binlog 转换而来的，具有精准、可控及快速的特点，给 DBA 运维工作带来非常大的便利性。对于 DDL 的执行，也会生成相应的回滚语句，不过只涉及语句级的回滚，而不关心数据。
- 支持打印增删改查语句的语法树功能，可以将具备很高门槛的 MySQL 语句的分析工作转换为老少皆知的 JSON 格式，这无疑降低了使用 SQL 语句的门槛，并且使得定制一些个性化规则成为了可能。
- 支持使用第三方工具 pt-online-schema-change 来在线改表的功能。这使我们在改大表时，与简单地修改一条记录的操作无异，并且可以查看详细的执行进度。简单、统一为运维带来了福音。
- 所有支持的这些功能，只需要通过提交时选择不同选项即可完成，整个过程对提交者来说是透明的。所有不同选项对应的功能，Inception 都会帮你自动完成，上层只需要使用它们即可。
- 所有交互方式都是以一问一答的形式来完成的。接口方面，就是我们熟悉的 MySQL 协议，直接使用即可。Inception 返回消息时，都是以结果集的方式呈现的，这使得我们的自动化运维程序实现起来更加方便、友好。

45 Inception 安装与使用

Inception 现在已经是一个被广泛使用并深受好评的开源软件，可以在 Github 上看到其活跃程度。在目前所处的 MySQL 大环境下，Inception 受到广泛欢迎的很大原因是，它支持的接口使用方式非常友好，并且非常通用，不局限于一个公司或某种应用场景下定制化的使用方式，而是在一种普遍存在并有着共性需求的基础上开发的。基于此，Inception 很符合大众口味，并且可以实实在在地解决非常多的问题，这样的产品，不会不受欢迎。

本章就从源码下载到编译安装，再到初步的简单使用方面进行介绍，让你可以初步认识到 Inception 的样子，认识 Inception 的征程也从现在开始。

下载和编译

这一节，先来讲述怎么安装 Inception。Inception 的源代码可以在 Github 上直接下载，开源 GitHub 地址为：<https://github.com/mysql-inception/inception.git>。

为了方便 Inception 的发展，并且期待每一个人都可以为 Inception 贡献自己的一份力量，这里推荐大家使用 git clone 的方式，或者用 fork 的方式获取源代码。这样，可以通过 Git 的管理及 Github 的平台，为 Inception 提交新的功能或修补一些错误等。更重要的是，Inception 代码会一直更新维护下去，通过这样的管理方式，也可以在更新之后，第一时间直接将最新的功能拉取到本地，并且测试上线。

使用 git clone 方式获取源码的操作如下。

```
git clone https://github.com/mysql-inception/inception.git
```

源码下载完成之后,接下来的工作就是编译了。因为 Inception 来源于对 MySQL 源码的修改,基本的一些操作和 MySQL 是一样的,所以还是通过 CMake 来配置编译环境,然后通过 Makefile 来编译生成可执行文件。但以 Centos 为例,这个过程需要依赖一些包才可以编译成功,依赖的包有如下五个。

- bison: 用来编译语法文件 (.yy)。如果 yum 源上面没有这个软件,则可以自行下载源码来安装,对应的网址为<http://ftp.gnu.org/gnu/bison/>。版本最好是 2.6 之前的,最新版本可能会遇到问题。下载之后,需要自己编译源码来安装,具体安装方法可以参照网上的一些说明。
- cmake: yum install cmake, 版本最好用 2.8.x 系列的,最新的不能保证不出问题。
- ncurses: yum install ncurses-devel.x86_64。
- openssl: yum install openssl-devel.x86_64。
- g++/gcc: yum install gcc-c++.x86_64 libgcc.x86_64 gcc.x86_64。

在将上面提到的这些包安装完成之后,基本编译就不会有问题了,简单步骤如下。

1. 进入 inception 源代码目录,创建一个编译目录,执行 mkdir debug, 这样编译生成的中间文件及可执行文件都会放到 debug 目录下,不会影响整个 inception 目录。需要重新编译时,只需要清空 debug 目录即可。
2. 进入 debug 目录,执行命令如下。

```
cmake .. -DWITH_DEBUG=OFF \
-DWITH_ZLIB=system \
-DCMAKE_INSTALL_PREFIX=/var/inception \
-DWITH_SSL=bundled \
-DCMAKE_BUILD_TYPE=RELEASE \
-DWITH_ZLIB=bundled
```

命令中的选项 DCMAKE_INSTALL_PREFIX, 是用来控制编译之后安装 Inception 时的目标位置的,这里指定了 /var/inception。而选项 DWITH_DEBUG 控制的是要不要编译为 DEBUG 版本,如果有调试需求,可以将这个参数设置为 yes。命令执行完成之后,在最后两行如果出现如下信息,则说明配置完成。

```
-- Generating done
-- Build files have been written to: /data/workspace/inception/xxx
```

3. 配置完成之后,就可以编译了,在当前目录下直接执行 make install 即可。执行完成之后,在目录 /var/inception 下面,就应该可以看到其安装完成的样子了。

现在可以进入目录/var/inception, 看到生成的所有目录及文件。不过这里需要注意的是, 因为 Inception 来源于 MySQL 源码, 所以这里除了 bin 目录下的 Inception 可执行文件之外, 其他的目录及文件都是不需要的, 可以直接忽略。

找到可执行文件 Inception 之后, 说明我们编译成功, 可以开始使用了。

为了更加方便地服务于大众, 本人已经将上面的操作做好了一键式的脚本, 在 inception 目录下, 文件名为 inception_build.sh。可以简单看一下脚本内容, 过程是一样的, 只不过有些选项内容不太一样, 这一点不用太关注, 使用方法如下。

```
sh inception_build.sh
```

会输出使用方法, 实际上只需要执行如下命令即可。

```
inception_build.sh debug [Xcode]
```

命令最后面的平台选项是可选的。如果不指定, 就是 Linux 平台; 如果指定为 Xcode, 后面就指定 Xcode。debug 是编译的目录, 编译之后, 所有的生成文件都在这个目录下, 包括可执行文件 Inception。可执行文件的目录路径为 debug/sql/Debug/ (不同平台有可能不同)。

在执行这个脚本时, 指定一个存在的编译目录时, 经常会出现如下的错误信息。

```
building project in xxxx
make: *** No rule to make target `install'. Stop.
```

这是因为, CMake 还没有完全配置好, 脚本便发现当前的目录已经存在了, 所以就不会再次重新做 CMake 配置, 而是直接去做 make install, 而此时的 Makefile 是不存在的, 因此报出这样的问题。解决的方法非常简单, 每次出错之后, 只需要把编译目录删除掉, 重新执行编译脚本即可。

顺便强调一下, 编译 Inception 和编译 MySQL 源码是一样的, 如果遇到了其他问题, 或者有不太了解的, 可以先在网上看一下关于 MySQL 源码的编译, 我想遇到的问题就都可以解决了。

启动配置

编译完成之后, 就是启动配置并使用了。只需要一个配置文件即可启动, 文件名为 inc.cnf, 如下。

```
[inception]
general_log=1
general_log_file=/var/inception/log/inception.log
port=6669
```

```
socket=/自己的目录, 请自行修改/inc.socket
character-set-client-handshake=0
character-set-server=utf8
inception_remote_system_password=root
inception_remote_system_user=zhufeng
inception_remote_backup_port=3306
inception_remote_backup_host=127.0.0.1
inception_support_charset=utf8mb4
inception_enable_nullable=0
inception_check_primary_key=1
inception_check_column_comment=1
inception_check_table_comment=1
inception_osc_min_table_size=1
inception_osc_bin_dir=/data/temp
inception_osc_chunk_time=0.1
inception_enable_blob_type=1
inception_check_column_default_value=1
```

上面这些参数的配置都是本人随意举例而已, 具体每个参数的意义, 请参照第 49 章。

Inception 可执行文件可以在编译目录下通过 `find` 命令找到, 编译目录就是在执行 `inception_build.sh` 脚本时指定的目录, 或在自行设置的目录 `/var/inception/bin` 下找到的。

到了启动时间了, 和 MySQL 类似, 启动有两种方式, 如下。


- `/var/inception/bin/Inception --defaults-file=inc.cnf`。
- `/var/inception/bin/Inception --port=6669`。

第二种方式是只指定一个端口, 其他参数都是默认值; 而第一种方式是在配置文件中可以指定很多参数, 按照自己喜欢的规则来配置。

启动成功之后, 可以通过 MySQL 客户端简单试一下看, 如下。

```
 mysql -uroot -h127.0.0.1 -P6669
```

登录上去之后, 再执行一条命令, 如下。

```
 inception get variables;
```

输出了所有变量, 恭喜你, 已经启动成功了。具体的使用命令等内容会在 51 章中具体讲述。

线上配置需求

对线上配置的需求如下。

- 线上服务器必须要打开 Binlog，在启动时需要设置参数 `log_bin`、`log_bin_index` 等关于 Binlog 的参数。不然不会备份及生成回滚语句，因为 Inception 的回滚语句生成是通过解析 Binlog 来做的。
- 参数 `binlog_format` 必须要设置为 `mixed` 或 `row` 模式，通过语句 `set global binlog_format=mixed/row` 来设置。如果是 `statement` 模式，则不做备份及回滚语句的生成。考虑到设置为 `statement` 时可能存在特别的原因，所以在 Inception 执行语句时，也不会将对应的会话级参数 `binlog_format` 改为 `ROW` 模式了，以免造成不可预知的影响。
- 参数 `server_id` 必须要设置为非 0 非 1，通过语句 `set global server_id=server_id;` 来设置，不然在备份时会报错。
- 线上服务器一定要有指定用户名的权限，这是在语句前面的注释中指定的，做什么操作就要有什么权限，否则还是会报错，如果需要执行的功能，则要有线上执行语句的权限，比如 DDL 及 DML。如果要执行 `inception show` 等远程命令，则有些语句是需要特殊权限的，且这些权限是需要授予的。关于权限这个问题，因为一般 Inception 是运行在一台固定机器上面的，那么在选项中指定的用户名和密码实际上是 Inception 机器对线上数据库访问的权限，所以建议在使用过程中，使用专门固定的账号来让 Inception 使用，最好是一个只读一个可写，这样在执行时用可写，审核/查看线上状态或者表库结果时用只读，会更安全。建议权限要包括 `SELECT`、`INSERT`、`UPDATE`、`DELETE`、`CREATE`、`DROP`、`PROCESS`、`ALTER`、`SUPER`、`REPLICATION SLAVE`、`REPLICATION CLIENT`、`TRIGGER`。
- 对涉及的表的要求是需要有主键。如果没有主键，则 Inception 会选择不生成回滚语句。原因很明显，就是对应的回滚语句的 `WHERE` 条件没法指定，指定一个全表的列也没有意义，所以选择不生成。
- Binog 参数 `binlog_row_image` 需要设置为 `full`，因为在备份生成回滚语句时都需要解析 Binlog，需要拿到记录的完整信息，不然备份不成功。

需要额外注意的点

在执行时，不能将 DML 语句及 DDL 语句放在一起执行，否则会由于表结构的变化而导致备份解析 Binlog 时出现不可预知的错误，如果要同时执行 DML 及 DDL，则需要分开多个语句块来执行，如果执行时一个语句块中同时包含了这二者，则 Inception 会报错，不会去执行。

使用方法

Inception 本质上是一个服务程序，类似于 `mysqld`。它应该有自己的一套友好的使用方式，必须要具备简单、高效、易用等特性。为了让 Inception 具有这些特点，在设计之初就规定了它的使用方式，如下所述。

Inception 对语句进行审核时,必须要告诉 Inception 这些语句对应的数据库地址、数据库端口及 Inception 连接数据库时使用的用户名、密码等信息,而不能简单地只是执行一条 SQL 语句,所以必须要通过某种方式将这些信息传达给 Inception。而我们选择的方式是,为了不影响语句的意义,将这些必要的信息都以注释的方式放在语句最前面,也就是说所有这些信息都是被 “/* */” 括起来的,每一个参数都是通过分号来分隔,类似的方式如下。

```
/*--user=username;--password=xxxx;--host=127.0.0.1;--port=3306;*/
```

当然,支持的参数不止是这几个,后面还会介绍一些其他的参数。

Inception 要做的是一个语句块的审核,而不是一次只对一条语句进制审核,所以需要引入一个规则,将要执行的语句包围起来。Inception 规定,在语句最开始的位置,要加上 `inception_magic_start`; 语句,在执行语句块的最后加上 `inception_magic_commit`; 语句,这 2 条语句在 Inception 中都是合法的、具有标记性质的、可被正确解析的“SQL”语句。被 “/* */” 括起来的所有需要审核或执行的语句都必须要在每条语句之后加上分号,其实就是批量执行 SQL 语句。(包括 `use database` 语句之后也要加分号,这点与 MySQL 客户端不同,不然存在语法错误。)

具体执行时,在没有解析到 `inception_magic_start` 之前,如果发现了需要执行的其他语句,则直接报错,因为在规则中, `inception_magic_start` 是强制存在的。而如果在执行的语句块最后没有出现 `inception_magic_commit`,则直接报错,不会做任何操作。在前面的注释部分,需要指定一些操作的选项,包括线上用户名、密码、数据库地址、检查/执行等,如下是一个简单的例子。

```
/*--user=zhufeng;--password=xxxxxxxxxxx;--host=xxxxxxxxxxx;
--enable-check;--port=3456;*/
inception_magic_start;
use mysql;
CREATE TABLE adaptive_office(id int);
inception_magic_commit;
```

那么上面这一段就是一批可以正常执行的 SQL 语句,但在 Inception 看来,这是一条语句,必须一次性提交给 Inception,但是在 Inception 内部,会对该段语句一条条地分别做审核操作。虽然 Inception 来源于 MySQL,但毕竟不是 MySQL,所以从思维上还是要有区别。所以目前的审核执行操作,只支持通过 C/C++ 接口、Python 及 PHP 等 MySQL 接口来对 Inception 进行访问,这一段必须是一次性地通过执行接口提交给 Inception。在处理完成之后,Inception 会返回一个结果集,来告诉我们这些语句中存在什么错误,或者是完全正常等。

这样的使用方式很容易理解,因为 Inception 作为自动化运维工具,必须是用程序去处理,而不是手工去 MySQL 客户端中进行操作。而 MySQL 客户端又有其自己的操作特点,比如它

会自己将所有语句通过分号分隔开来，然后一条条地向 MySQL Server 发送，而这样的操作方式是不适用于 Inception 的，所以都会报错，因为不符合 Inception 的协议。

所以，请不要将上面的 SQL 语句块放到 MySQL 客户端中执行。需要强调的是，Inception 是一个自动化运维工具，如果使用交互式的命令行来使用的话没有意义，只能通过写程序来访问 Inception 服务器。

关于为什么只支持通过 C/C++ 接口、Python 及 PHP 等 MySQL 接口来对 Inception 访问的问题，这里简单说明一下，因为 Inception 毕竟不是 MySQL，而很多人所熟悉并期待的 Java 客户端在发起一个连接时会做非常多的操作，比如获取变量、设置环境等，而很多操作 Inception 都是不支持的（因为不是 MySQL），Java 客户端在遇到返回错误的时候，就叫停了，所以 Inception 不支持 Java 客户端。除此之外，一些轻量级的客户端应该都是没有问题的，上面列出来的只是一些比较常用的，大家可以尽展其才，做更多的尝试。


可以通过 MySQL 客户端来执行的，只有 Inception 命令，请参考 51 章。

举例说明


通过上面的讲述，现在已经了解了 Inception 的基本用法。下面通过一个简单的示例，来更形象地说明如何来使用 Inception。为了简单，采用的是 Python 的 MySQL 客户端。

为了简单操作，避免 Python 字符串的转义问题，最好将要审核的 SQL 文件和 Python 程序分开，这样的话，SQL 文件就可以随便改，只要是正确的 SQL 语句，就不会存在 Python 相关的问题。

SQL 文件，文件名为 sql，如下。

```
 /*--user=username;--password=password;--host=127.0.0.1;--execute=1;--port=3306;*/
inception_magic_start;
set names utf8;
use mysql;
CREATE TABLE adaptive_office(id int);
inception_magic_commit;
```

Python 程序如下。

```
 #!/usr/bin/python
# -*- coding: utf-8 -*-

import json
import os
import MySQLdb
```



```

from MySQLdb.constants.CLIENT import MULTI_STATEMENTS, MULTI_RESULTS

file_object = open('sql')
try:
    all_the_text = file_object.read( )
finally:
    file_object.close()

try:
    conn=MySQLdb.connect(host='127.0.0.1',user='',passwd='',db='',port=9998,
client_flag=MULTI_STATEMENTS|MULTI_RESULTS)
    cur=conn.cursor()
    ret=cur.execute(all_the_text)
    num_fields = len(cur.description)
    field_names = [i[0] for i in cur.description]
    result=cur.fetchall()
    cur.close()
    conn.close()
    print field_names
    for row in result:
        print row[0], "|", row[1], "|", row[2], "|", row[3], "|", row[4], "|",
row[5], "|", row[6], "|", row[7], "|", row[8], "|", row[9], "|", row[10]
except MySQLdb.Error,e:
    print "Mysql Error %d: %s" % (e.args[0], e.args[1])

```

执行这段程序之后，返回的结果如下。

```

13 ['ID', 'stage', 'errlevel', 'stagesstatus', 'errormessage', 'SQL', 'Affected_rows',
'sequence', 'backup_dbname', 'execute_time', 'sqlsha1']
1 | CHECKED | 0 | Audit completed | None | use mysql | 0 | '0_0_0' | None | 0 |
2 | CHECKED | 1 | Audit completed | Set engine to innodb for table 'adaptive_office'.
Set charset to one of 'utf8mb4' for table 'adaptive_office'.
Set comments for table 'adaptive_office'.
Column 'id' in table 'adaptive_office' have no comments.
Column 'id' in table 'adaptive_office' is not allowed to be nullable.
Set Default value for column 'id' in table 'adaptive_office'
Set a primary key for table 'adaptive_office'. | CREATE TABLE adaptive_office(id int)
| 0 | '0_0_1' | 127_0_0_1_3306_mysql | 0|

```

从返回结果可以看到每一行语句的审核及执行信息。最前面打印的是 `field_names`，表示 Inception 的返回结果集的列名信息，总共包括十列，上段代码的下面是每列对应的结果，因为只有两条语句，所以只有两行。从结果集第一个列看到只有序号为 1 和 2 的两行，而对于每一列的具体含义，会在第 50 章中讲到，这里只需要看清楚是什么内容即可。

需要注意的是，最后一个“|”后面其实是存储列 sqlsha1 的，但这里没有改表语句，所以都是空。关于这个信息，请参考第 50 章及第 52 章中“对 OSC 的支持”一节。

环境变量的设置

在提交 Inception 任务时，整个语句块被 Inception 认为是一条单独的 SQL 语句，在内部会逐条地拆开。那么，在分析的过程中，必然会有一些参数一直影响着所有语句的执行，这些参数被 Inception 称为环境变量。目前支持的环境变量有两种，分别是 `use db` 和 `set names charsetname` 两种语句。

环境变量，在 Inception 的执行过程中，会全程陪伴。如果执行错误了，那么在重新生成任务时，不仅是要把未执行的语句提取出来，还需要把这些环境变量的变迁过程按照顺序获取到，目的是要得到未执行语句前面的最后一个环境变量的语句，这也正是为什么 Inception 返回结果集中 stage 列的值存在 RERUN 情况的原因了。

如果它的值是 RERUN，这就表示当前语句是用来设置环境变量的，在开发自动化运维平台时，这是需要格外注意的。如果只是提取了未执行的语句，那是没有意义的，因为首先会丢失字符集设置（有时候会引起乱码），并且导致一些表找不到其对应的库，从而导致再次执行失败的问题。下面就分别来介绍一下两种设置环境变量语句的意义。

use db

在执行过程中，必然会找到每一个使用到的库、表等对象在线上环境中的对应关系，也就是里面包含了所有 SQL 语句对应的上下文环境，从前面的例子中也可以看出，里面通过 `use` 语句来指定了库名。

关于 `use database`，如果没有使用 `use` 语句，则必须要在使用表的同时指定库名，即 `database-name.tablename`，否则 Inception 会直接报错。如果通过 `use` 语句指定过一次数据库，那么在当前语句块中，后面可以不再指定，除非想要修改当前库的名称。而如果只是通过 `database-name.tablename` 方式指定了表的位置，则后面的操作不会受到前面的影响，也就是后面还需要明确指定库名。

对于上面两种不同的指定库名的方式，`use database` 是可以影响上下文环境的，而 `database-name.tablename` 的方式，只会影响语句自己。当然在 `use db1` 之后，如果想要执行类似 `insert db2.table1` 的语句，也是可以的，此时这条语句本身就会临时切换为 `db2` 库，而之后的语句如果没有明确指定库名的话，还会恢复到 `db1` 的环境中。

set names charsetname

对于我们熟悉的 `set names utf8;`, 可以在所有语句开头指定, 如果后面不再出现并且不做修改, 则所有语句都会以 `utf8` 的字符集来执行, 而如果在中间某一个位置重新指定了, 比如 `set names utf8mb4;`, 那么从这个设置开始, 后面所有的语句执行都会切换为 `utf8mb4` 的字符集。

46

支持选项

前面讲述了关于 Inception 的基本使用方法，在一些例子中已经看到了部分选项。不同的选项，对应的 Inception 功能也是不同的，相应的 Inception 行为与结果都会有所不同。那么，本章将会详细讲述 Inception 目前所支持的每一个选项及其各自的意义。

选项说明

选项是在 SQL 语句块前面以注释的方式一起传达给 Inception 的，它支持的选项很丰富，这里将详细介绍现在所支持的每个选项的使用方法及其意义。

参数名	是否需要参数	参数是否可选	功能描述
--host	是	否	需要操作的这部分语句块对应的数据库地址。指定方式可以是 IP 地址、机器名或 DNS 域名等，只要能唯一解析到这个机器即可
--port	是	否	与上面的--host 选项对应，指定机器名，必然要再指定一个 MySQL 实例的端口，那就通过--port 来完成

续表

参数名	是否需要参数	参数是否可选	功能描述
--user	是	否	已经确定了 MySQL 实例,那就需要指定如何连接这个实例,通过--user 指定用户名,且这个用户必须要在上面指定的后端 MySQL Server 上有相应的权限,在前面已经对需要什么权限做过说明
--password	是	否	已经确定了 MySQL 实例,那就需要指定如何连接这个实例,通过上面的--user 指定了用户名,那么 --password 所指定的就是这个用户名对应的密码,必须要以明文的方式来指定
--sleep	是	是	这个参数用来指定在执行完每一条语句后,暂停多少毫秒,这样可以适当控制对线上数据库的冲击,特别是针对大量写入的操作,单位为毫秒,最小值为 0,也就是不暂停,最大值为 100 秒,也就是 100000 毫秒。如果设置得超过 100000 毫秒,Inception 会自动将其设置为 100000 毫秒。这个参数可以和其他参数一起设置,但是只有在 --enable-execute 为 1 的情况下,才起作用
--enable-check	否	否	告诉 Inception 当前要做什么操作,是审核还是执行,这个参数与下面的--enable-execute 只能指定一个,功能如其名,--enable-check 就是告诉 Inception,当前的请求是要做审核操作,审核完成就返回结果集

续表

参数名	是否需要参数	参数是否可选	功能描述
<code>--enable-execute</code>	否	否	告诉 Inception 当前要做什么操作，是审核还是执行，这个参数与上面的 <code>--enable-check</code> 只能指定一个。如果指定的是选项 <code>--enable-execute</code> ，则 Inception 在执行前还会做一次实时的审核，这个审核和前面指定 <code>--enable-check</code> 时的审核基本是相同的，只是这次在审核完成之后，还会继续执行。因为相同的语句在不同的时间审核有可能会产生不同的审核结果（环境有可能变了），所以有必要再做一次审核。如果审核发现了错误（而不是警告），就不会被执行，此时会提前返回告知错误，如果审核时发现的是警告，并且没有指定 <code>--enable-ignore-warnings</code> （下面会介绍），则有警告也不会执行，而如果想跳过这些警告，则可以通过下面这个选项来设置
<code>--enable-ignore-warnings</code>	否	是	Inception 采取严格的分阶段处理方式，先对所有语句进行审核，审核完成之后，会执行所有语句，之后才会去做所有语句的备份操作。在三个阶段的过渡过程中，如果审核有问题则不会继续执行，此时如果人为确定想要跳过这些警告，也就是需要忽略它们，则可以选择这个参数，告诉 Inception 跳过这个警告的检查，继续执行

续表

参数名	是否需要参数	参数是否可选	功能描述
--enable-force	否	是	在使用 Inception 的时候，还经常会遇到一个问题，那就是批量导入数据时，执行某些语句时有可能会报出主键冲突的问题，而 DBA 可以确定的是，出现主键冲突不是问题，可以继续执行，那么此时就可以通过选项--enable-force 告诉 Inception，在执行过程中碰到一个错误时，可以先保存错误信息并继续执行下一条语句。这个参数要谨慎使用
--enable-remote-backup	否	是	Inception 支持备份并生成对应的回滚语句，这是默认的，但当有些影响行数很多且明确不需要回滚的时候，为了提高执行效率，可以指定在执行时不做备份，指定方式是通过 disable 来禁用它。这种方式是 MySQL 配置文件禁用某个选项时常用的方法，即--disable-remote-backup。关于备份的具体内容，在 47 章中会有专门介绍
--enable-split	否	是	这个参数用来拆分要执行的语句块。如果在语句块中存在对同一个表的 DDL 操作及 DML 操作，那么在分析 Binlog 来生成回滚语句时，由于表结构已经发生改变，会导致 inception 没有办法处理，所以使用这个参数将这些语句分成多批，然后再分别执行。这是在执行前必须要做的一个操作，不然可能会产生不可预知的错误。当然在执行前的最后一次审核中，如果检查到这样的混用情况，则会返回报错，而不是警告。这个参数指定之后，除了前面四个参数之外，其他参数都被忽略，也可以不指定

续表

参数名	是否需要参数	参数是否可选	功能描述
<code>--enable-query-print</code>	否	是	这个参数用来打印 SQL 语句在被 Inception 分析之后的执行树结构，以 JSON 的形式提供，目的是为了可以在 Inception 的基础上，对已经结构化的 (JSON) 语句做再次分析，可以对 Inception 内置支持的规则进行扩展，做个性化定制，比如使用到哪些列、哪些语句类型等信息。目前支持的语句类型有插入、删除、更新及查询，详情请参考 52 章中“打印语法树”一节

DDL 与 DML 语句分离

首先要说明的一点是，这一节是对上面所述选项“`--enable-split`”的进一步阐述。

Inception 支持将一段 SQL 语句按照语句之间相互不影响的原则把 DDL 和 DML 语句分开，也就是让相同表的 DDL 和 DML 语句不在同一个语句块中执行。

与其说这是一个功能，不如说这是一个必经流程，为什么我们要实现这样一个功能呢？原因是，如果在同一个块中同时执行 DDL 和 DML 的混合语句体，那么 Inception 对表的缓存结构在变化之后（因为表的修改导致的），就会导致不能正确地解析修改之前的 Binlog 记录，从而导致备份出问题，所以 Inception 采取了一个简单折中的办法来处理这个问题。

这个功能的用法是通过指定另一个选项来实现的，这个选项为 `--enable-split`。在指定这个选项之后，所有其他选项（除了指定线上地址的 4 个选项之外）都不起作用，只处理 split 选项。

这个功能的输出也是一个结果集，只是这个结果集只有 3 列，分别如下。

- ID：第一列是序号，表示当前行被分成第几个语句块，这个值是从 1 开始的。
- sql_statement：第二列表示的是可以在一起的所有 SQL 语句，都合在一起并且都以分号分开。
- ddlflag：第三列表示的是当前被拆分之后，可以一起执行的语句中，有没有 alter/drop table 语句，如果有则输出 1，否则输出 0。这主要是为了避免在 Inception 执行这两种语句时有可能带来的危险，这样输出之后，可以为上层提供更友好的选择，这个更友好的选择可以作为一个提醒。

如果第一列返回的序号为 0, 那么此次结果集只会有一行, 并且这一行的第二列的值是出错信息, 也就是说这次操作是有错误的, 具体错误可以直接看到。是不是有错误发生, 只需要看序号值是不是为 0 就可以了。

经过这样的处理, 就可以放心地将每一行单独作为一个 Inception 任务提交了, 此时可以直接以 `--enable-execute` 类型的任务提交。

所以从这里可以看出, 对于执行一个任务而言, 一个完整的流程应该是如下这样。

- 流程一: `enable-check`。
- 流程二: `enable-split`。
- 流程三: `enable-execute(s)`。

流程三中最后的 `s` 的意思是, 被 `split` 处理过之后, 有可能需要分多次来执行。

知道这两种语句拆分的用法之后, 应该还会想知道在内部是如何拆分的。其实, 在知道拆分的目的之后, 拆分原则也应该猜到一二了。

这个拆分原则是, 通过分析, 按照在语句块中的出现顺序, 对库表及操作类型进行统计, 在扫描的过程中, 从某一个表中第一次出现 DDL 和 DML 共存情况的这条语句开始, 重新生成一个语句块, 而这条语句之前的内容就作为一个完整的语句块。产生新的语句块之后, 继续向后扫描, 直到处理完所有的语句为止, 扫描结束之后, 分成多少块, 那就是 `split` 的结果集中有几行。

很明显, 如果一个语句块中的所有语句涉及的表都不相同, 或者是涉及的不是同一个库下面的表, 又或者涉及了相同的表但都是 DDL 或 DML 等, 那么就都不会被拆开。经过 `split` 处理之后, 输出结果集其实还是只有一行。这有点类似一条流水线, 可以称之为“合格检查”, 同一个表的 DDL 和 DML 语句共存, 就是不合格, 除此之外, 都是“合格产品”。

在拆分过程中, 还有一点需要注意, 因为其中会涉及环境变量的问题, 所以拆分过程不能简单地把所有语句机械地从物理上分开, 每个语句还是要依赖于环境变量, 比如 `db` 信息、`charset` 信息等。所以 Inception 在拆分过程的每一段中, 都会将它们的环境变量继承过来, 保证逻辑上不会有问题。

小技巧

可以看到, 上面所说的参数名除了前面四个连接选项之外, 都在前面加了 `enable`。实际上, 这些参数的写法与 MySQL 配置参数的写法是相同的, 比如 `--enable-check`, 这个参数的名字只是简单的 `check`, 可以写成 `--check=1`, 也可以写成 `--disable-check`, 都是比较灵活的, 其他的也都是一个道理。

从上面列出的选项来看，做任何操作时，有四个选项是必须要有的，分别是--host、--port、--user 和--password。

而有一些选项之间是互斥的，也就是不可以同时出现的。这些选项分别是--enable-check、--enable-execute、--enable-split 和--enable-query-print。这也说明，Inception 目前从选项上来说支持这四个功能。

还有一些选项，是用来辅助执行的，分别如下。

- --sleep，执行间隔，辅助的选项是--enable-execute。
- --enable-ignore-warnings，忽略警告，辅助的选项是--enable-execute。
- --enable-force，跳过错误，辅助的选项是--enable-execute。
- --enable-remote-backup，备份，辅助的选项是--enable-execute。

这样归类之后，对 Inception 的功能应该就比较清楚了，也可以知道在什么情况下该用什么选项了。

47

Inception 的备份回滚

备份存储架构

46 章中已经提到，Inception 在做 DML 操作时，具有备份功能。它会将所有当前语句修改的行对应生成回滚语句并备份下来，同时也会将所有操作的任务备份下来，一起存储到一个指定的库中，这些库的指定需要通过 4 个参数，分别如下。

- `inception_remote_backup_host`: 指定远程备份 MySQL 实例的地址。
- `inception_remote_backup_port`: 指定远程备份 MySQL 实例的端口。
- `inception_remote_system_user`: 备份时，连接上面指定的 MySQL 实例时所需要的用户名，这个用户需要有相应的权限，一般包括 CREATE、INSERT 及 SELECT 权限。
- `inception_remote_system_password`: 备份时，连接备份库时所需要的用户对应的密码。

这些参数可以直接在命令行中指定，也可以用 `--defaults-file` 指定一个 `inc.cnf` 文件，一经指定，再不能更改，除非将服务器关闭，修改之后再次启动。文件中指定这几个参数的值，用法与 MySQL 是相同的。

如果没有设置上面四个参数而又想要使用备份功能，那么在提交任务时，Inception 就会报错，错误信息是 `Invalid remote backup information`。

Inception 默认是开启备份功能的。为了使备份成为一个会话级的可选功能，有一个辅助选项可以设置，即前面讲述关于 Inception 支持选项的内容中所说到的 `--disable-remote-backup`。

备份信息及回滚语句,在备份机器上的存储与线上被修改的数据库是一一对应的。但因为线上机器会很多,而备份机器只有一台,所以为了防止在备份数据库实例中存在库名冲突的问题,备份机器的库名是将线上机器 IP 地址的点换成下划线,再加上端口号及库名,由这三部分组成。这三部分也是通过下划线连接起来的。例如 192_168_1_1_3310_inceptiondb。下面用 inceptiondb 作为线上库名来举例。

对于一个备份库,它所包含的表与相应的线上表是一一对应的,也就是说线上库 inceptiondb 中有什么表,在备份库 192_168_1_1_3310_inceptiondb 中就有什麼表,表名也完全相同,不同的只是表结构,它用来存储所有对这个表修改的回滚语句,表结构如下。

```
CREATE TABLE `tablename` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `rollback_statement` mediumtext,
  `opid_time` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8
```

每一列的解释如下。

- **rollback_statement**: 该列存储的是当某一行被修改后,生成的对应的回滚语句。因为 Binlog 是 ROW 模式的,所以不管是什么语句,产生的回滚语句都是针对一行的。同时,如果一条语句的执行影响了多行,那么这里就会有多条回滚语句,但它们对应的是同一条 SQL 语句。
- **opid_time**: 这一列存储的是被执行的 SQL 语句在执行时获取的一个序列号,这个序列号由 3 部分组成: timestamp (int 值,是语句被执行的时间点)、线上服务器执行时所产生的 thread_id 及当前这条语句在所有被执行语句块中的一个序号。产生结果类似下面的样子: 1413347135_136_3,这个序列号在指定的一个备份库中是唯一的。针对同一条语句影响多行的情况,在产生的多行回滚语句中,该列的值都是相同的,这样就可以找到一条语句对应的所有被影响数据的回滚语句。因为这个值是唯一的,所以可以通过这个值在备份库中找到某一条 SQL 语句对应的所有回滚语句。

除了与原库中的表一一对应之外,每个备份库中还有另外一个表: `$_$Inception_backup_information$_$`,这是其表名,有些难看,主要是为了防止与原线上库中的表名发生冲突。该表用来记录所有对当前库的操作,它是为该库中的所有表服务的,对线上这个库中所有的表的操作,都会被存储在这里面。该表的结构如下。

```
CREATE TABLE `$_$Inception_backup_information$_$` (
  `opid_time` varchar(50) DEFAULT NULL,
  `start_binlog_file` varchar(512) DEFAULT NULL,
  `start_binlog_pos` int(11) DEFAULT NULL,
```

```

`end_binlog_file` varchar(512) DEFAULT NULL,
`end_binlog_pos` int(11) DEFAULT NULL,
`sql_statement` text,
`host` varchar(64) DEFAULT NULL,
`dbname` varchar(64) DEFAULT NULL,
`tablename` varchar(64) DEFAULT NULL,
`port` int(11) DEFAULT NULL,
`time` timestamp NULL DEFAULT NULL,
`type` varchar(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8

```

从而可以知道，线上库表结构与备份库表结构的对应关系如图 47.1 所示。

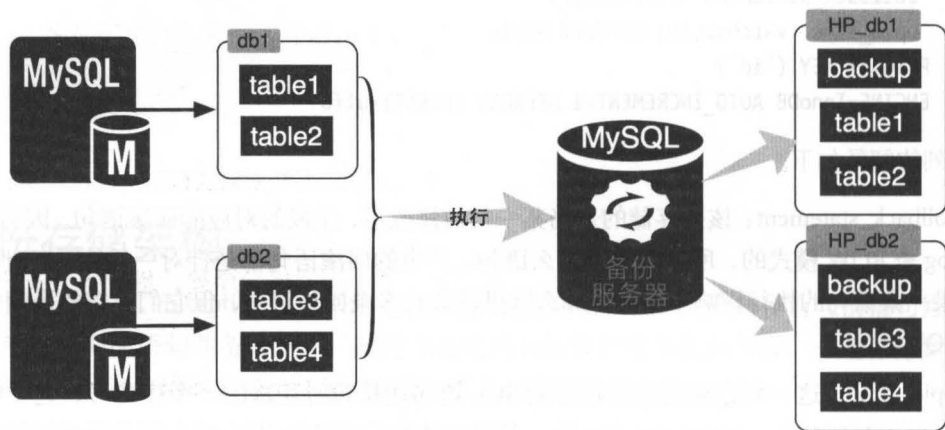


图 47.1



注意：图 47.1 备份实例中的每一个库中的表 backup，实际上就是表 `$_Inception_backup_information$_`。

下面详细解释一下每一列的作用及意义，表中的每一行，对应在线上执行的一条实际的 SQL 语句。

- `opid_time`：该列与上面备份表中的列 `opid_time` 是一一对应的，因为这个表中每一行对应的是在线上执行的一条实际的 SQL 语句，所以 `opid_time` 的意义就是用来唯一表示这条语句的。`opid_time` 从各个备份表中查找这条语句对应的回滚语句，是一对多的关系。
- `start_binlog_file`：该列从名字中可以看出，表示的是执行这条语句前 Binlog 所在位置的文件名。当然这个值不一定准确，因为这是在执行语句前通过 `show master status;` 语句来获取的，在数据库并发比较高的情况下，这个值一般都不是当前语句的 Binlog 开始的

准确位置，这个位置只能是这条语句产生 Binlog 前面的某个位置。同理，下面三个位置信息也是一样。

- `start_binlog_pos`: 该列与上面的列对应，表示的是上面指定文件的位置信息。
- `end_binlog_file`: 该列表示的是执行当前语句之后，Binlog 所在的文件名。
- `end_binlog_pos`: 该列与上面的列对应，它表示执行完成之后，Binlog 在文件 `end_binlog_file` 中的偏移位置。
- `sql_statement`: 该列存储的是当前这个被执行的 SQL 语句。
- `host`: 表示在线上的哪个数据库实例中执行了这条语句。存储的是当前提交语句时在注释中的指定地址。
- `dbname`: 表示执行当前语句时所处的环境变量，指的是数据库名。
- `tablename`: 表示当前语句影响的表的表名，可以通过这个名字对应到备份表名。
- `port`: 与 `host` 对应，表示执行时数据库的端口是什么。
- `time`: 表示当前语句的执行时间。
- `type`: 表示操作类型，现在只支持 INSERT、UPDATE、DELETE、CREATEDB、CREATETABLE、ALTERTABLE、DROPTABLE 等类型。

依照现在备份及回滚的实现方案，如果已经知道一条语句的执行序列，想拿到这条语句的回滚语句，那么要执行的 SQL 语句如下。

```
select rollback_statement from 192_168_1_1_3310_inceptiondb.inception_test
where opid_time = '1413347135_136_3';
```

上面语句查出来的只是针对一个语句块中某一条语句的回滚语句，如果想要得到整个语句块的回滚语句，还需要在此基础上做二次开发。首先根据结果集中返回的每一条语句对应的 `opid_time` 值，找到所有的回滚语句，然后分不同语句按照执行顺序的倒序排列，再去线上执行，这样才可以保证回滚正确。不过为了安全起见，可以把这些回滚语句放到一个事务中进行处理，如果事务太大，可以按照 `opid_time` 来分组，并针对每一个 `opid_time` 使用一个事务来执行，保证同一条语句的回滚是原子性的。回滚方式如图 47.2 所示。

关于 DDL 的回滚，其实很难做得完美，因为涉及数据的大批量更改，并且 Binlog 是语句模式的，所以很难处理。但是，在 Inception 中采取的策略是只处理定义，不处理数据。现在 DDL 操作的回滚只包括 CREATE TABLE、ALTER TABLE、DROP TABLE，其他类型的操作不支持，并且也没有太大意义，在实际使用中如果有需求再考虑如何实现。

每条 DDL 语句算一个操作，同样地，这个操作也会存储在表 `$_$Inception_backup_information$$` 中，关于 Binlog 的一些列没有实际意义，对应的 `opid_time` 与 DML 是相同的。而同时，在被操作表中存储了回滚语句，其中的列 `opid_time` 与上面表中的这个列关联。一个 DDL

操作（不管其中做了多少事），对应的回滚语句也是一条语句，例如下面这个例子。原来的表是如下这样的。

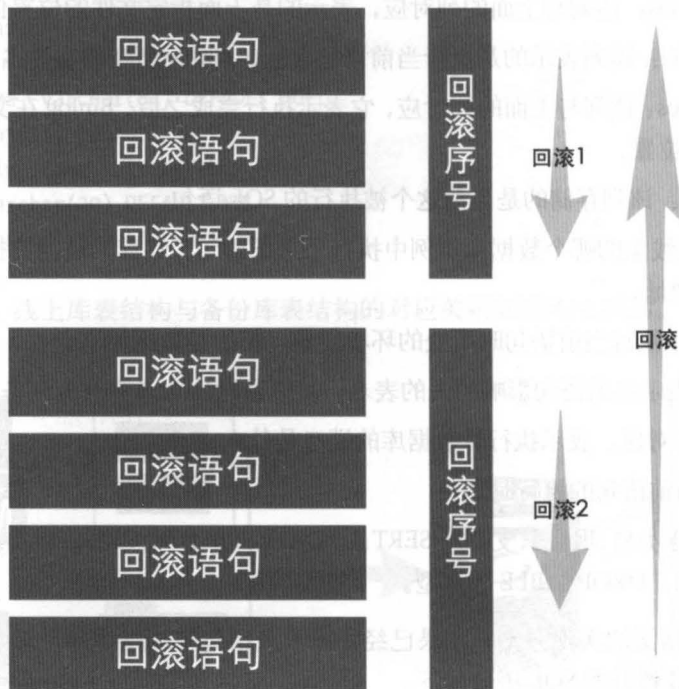


图 47.2

```
CREATE TABLE myinfo (
  id int(11) NOT NULL AUTO_INCREMENT COMMENT 'id',
  name varchar(10) NOT NULL DEFAULT 'baabb' COMMENT 'agedddd',
  age2 int(11) NOT NULL COMMENT 'agddd',
  age int(11) NOT NULL COMMENT 'age',
  PRIMARY KEY (id),
  KEY idx_name (id,name)
) ENGINE=InnoDB AUTO_INCREMENT=2150 DEFAULT CHARSET=utf8 COMMENT=' t' ;
```

要执行的 alter 表语句如下。

```
alter table myinfo
rename to myinfo1,
add column age3 int not null comment 'age',
add index idx_age2(age2),
drop column age;
```

产生的回滚语句如下。

```
ALTER TABLE inception.myinfo1
DROP COLUMN age3,
DROP INDEX idx_age2,
ADD COLUMN age int(11) NOT NULL COMMENT 'age',
RENAME TO inception.myinfo;
```

备份所需条件

Inception 支持了 DDL 及 DML 执行的备份回滚，但要想正确地使用并产生相应的预期结果，还需要注意一些细节。下面是它要满足的一些条件。

- 被修改表需要有主键：执行时，被影响的表如果没有主键的话，就不会做备份了。这样更简单并且备份时间及数据都会少一点，不然回滚语句的 WHERE 条件就会将所有列写进去，这样会影响性能且没有太大意义，所以在 WHERE 条件中，只需要主键即可。
- 参数 server_id 必须要设置为非 0 及非 1，通过语句 set global server_id=server_id; 来设置，不然在备份时会报错。因为在获取 Binlog 时，需要通过 server_id 在主库上注册 Inception 的信息。
- 线上服务器必须要打开 Binlog，在启动时需要设置参数 log_bin、log_bin_index 等关于 Binlog 的参数。不然不会备份及生成回滚语句，因为 Inception 的生成回滚语句是通过解析 Binlog 来做的。
- 参数 binlog_format 必须要设置为 mixed 或者 row 模式，通过语句 set global binlog_format=mixed/row 来设置。如果是 statement 模式，则不做备份及回滚语句的生成。这考虑到设置为 statement 时可能存在特别的原因，所以 Inception 执行语句时，也不会将对应的会话级参数 binlog_format 改为 ROW 模式了，以免造成不可预知的影响。
- 备份相关的四个参数需要设置好，并且对应的用户在备份数据库实例中有相应的权限。

48

审核规范

今天，业界已经基本普遍认同，在 MySQL 的应用实践中需要对应用程序使用的 SQL 语句进行审核，以确保生产环境中的 MySQL 数据库服务的正常稳定运行。实际上，对 SQL 语句审核还会带来其他附加的收益，比如语句风格的统一与规范、SQL 的标准化、SQL 变更的审计等。建立审核标准是一项非常有意义的工作，而之前的人工审核，针对标准这个问题其实是很吃力的，标准越多，需要记忆和评估的点就越多，DBA 越累，开发也越累。Inception 出现之后，它是不怕标准多的。只要 DBA 定义了更好地管理数据库的规则，Inception 通过编程实现之后就可以很好地执行，并且返回审核结果。

这一章主要介绍当前 Inception 在审核时，是使用什么样的规则来做的，以及哪些规则是可以配置的，哪些规则是不可以配置的。这样，针对不同的部门或公司，可以定制不同的规则，配置参数可以参考第 49 章。

支持的语句类型

- use db: 此时会检查这个库是否存在，需要连接到线上服务器来判断。Inception 分析到这个语句的时候，也是变更环境变量的时候，这条语句之后的所有语句执行都会在这个库下面，除非重新设置或语句中明确指定了库名。
- set option: 现在只支持 set names charset，设置其他变量时都会报错不支持。这个操作也是一个环境变量设置语句。

- 创建数据库语句：创建数据库，会在 Inception 内部对这个新数据库做缓存，这样就可以做到在一个语句块中，新建数据库并且在当前语句块中直接使用（而不需要先在线上创建这个数据库），给实际应用带来了非常大的便利性。
- 插入语句：包括多值插入操作，比如 `insert t1 values(c11, c21), (c12, c22)` 类似的语句。
- 查询插入语句：这种类型也属于插入语句，只不过后面跟着的是一个 `SELECT` 查询，和简单的插入有区别，比如预计影响行数的处理就不同，简单的 `INSERT` 可以直接计算出精确的影响行数，而查询插入就做不到这一点。
- 删除语句：包括多表删除语句，主要分析影响行数，表达式的合法性等方面的问题。
- 更新语句：包括多表更新语句，主要分析影响行数，表达式的合法性等方面的问题。
- 创建表语句：包括新表创建及根据一个已经存在的表做 `LIKE` 创建。创建表之后，也会被 Inception 做本地缓存，这样在同一个语句块中，创建之后马上就可以使用，给业务开发同学及 DBA 带来了很大的便利性。
- 删除表语句：根据实际需求，删除也是有必要的，不过要谨慎行事。
- 修改表语句：改表是常用的需求，这里有可能会使用到我们熟悉的 `OSC`，其支持丰富的改表类型。
- `Truncate` 表语句：和删表的道理一样。
- Inception 命令集语句。

公共检查项

- 使用到的库、表、列对象是否存在。如果不存在，则会报错并将错误信息加入到对应语句的结果集列中。这种类型的错误，是真的错误，而不是警告，结果集中的 `errlevel` 为 2。
- 检查创建对象的名字的合法性。如果名字长度大于 64 个字节，就会报错。包括库名、表名、索引名、列名等。
- 检查标识符的合法性。如果发现名字中存在除数字、字母、下划线之外的字符，则会报错 `Identifier "invalidname" is invalid, valid options: [a-z,A-Z,0-9,_,]`。
- 检查 `SET OPTION` 语句类型。目前只支持 `set names charsetname` 语句，类型为错误，不可配置。
- 检查创建表、库对象时使用的字符集是否是参数中所定义的，类型为警告，可配置。
- 检查语句类型。如果是 `Create Index` 语句，或是 `Rename Table` 语句，又或是 `Drop Index` 语句，都建议使用 `Alter Table` 语句，类型为警告，不可配置。

- 检查同一个表是否存在 DDL 与 DML 共存的问题，类型为错误，不可配置。虽然在选项中支持--enable-split 来解决这个问题，但是为了防止没有做这个操作而出现问题，还是要从安全性层面上检查一次。
- 检查语句中有没有出现 MySQL 关键字，类型为警告，可配置。
- 检查影响行数是否超过某一个设定的值，类型为警告，可配置。

插入语句检查项

- 插入指定的列列表中，同一个列不能出现多次，类型为错误，不可配置。
- 必须指定插入列表，也就是要对哪几个列指定插入值，如 insert into t (id, id2) values(...), 类型为警告，可配置。
- 插入列列表与值列表个数相同，二者的个数需要相同，如果没有指定列列表（因为可配置），则值列表长度要与表列数相同。类型为错误，不可配置。
- 检查 NOT NULL 的列，插入的值是不是 NULL，类型为错误，不可配置。
- 检查查询插入时，查询的列个数，与指定列的列表元素个数是否相同，类型为错误，不可配置。
- 检查查询插入语句中的 SELECT 子句，有没有 WHERE 条件，类型为警告，可配置。
- 检查查询插入语句中的 SELECT 子句，是不是“SELECT *”，类型为警告，可配置。
- 检查查询插入语句中的 SELECT 子句，有没有 LIMIT 子句，类型为警告，可配置。
- 检查查询插入语句中的 SELECT 子句中的 ORDER BY 子句，排列类是否为 RAND()，类型为警告，可配置。

更新、删除语句检查项

- 检查更新删除语句中的 SELECT 子句有没有 WHERE 条件，类型为警告，可配置。
- 检查更新删除语句中的 SELECT 子句有没有 LIMIT 子句，类型为警告，可配置。
- 检查更新删除语句中的 SELECT 子句中的 ORDER BY 子句，排列类是否为 RAND()，类型为警告，可配置。

表属性检查项

- 检查表的存储引擎是不是 InnoDB，有参数可以配置。
- 检查表的字符集是不是与参数配置匹配，有参数可以设置支持的字符集。

- 检查表是不是有注释，有参数可以设置。
- 检查表是不是分区表，有参数可以配置。
- 检查 LIKE 建表时，参照的对象表是不是存在，类型为错误，不可配置。
- 如果建立的是临时表，则必须要以 tmp 为前缀，类型为错误，不可配置。
- 检查创建的列，是否存在重名的列，类型为错误，不可配置。
- 检查新表有没有主键，类型为警告，可配置。
- 检查自增列初始值是不是为 1，不是则报警告，可配置。
- 检查表中是不是存在多个主键，类型为错误，不可配置。

列属性检查项

- 检查 BLOB 列是不是设置了默认值，如果是则报警告，无参数可设置。
- 检查列类型是不是 DATETIME，并且其默认值为 NOW()，如果是则报错，类型为错误，不可配置。
- 检查列设置是不是 NOT NULL 属性，或者为主键列，但其默认值被设置为 NULL，如果是则报错，类型为错误，不可配置。
- 检查自增列是不是设置了默认值，如果是则报错，类型为错误，不可配置。
- 检查时间类型默认值的合法性，如果与对应的字段类型不匹配，则报错，类型为错误，不可配置。关于原则，可以参照 MySQL 在设置为 strict mode 时，对应的时间类型如何使用的原则。
- 检查列数据类型是不是 SET，或者是 ENUM，抑或是 BIT，类型为警告，可配置。
- 检查列有没有注释，类型为警告，可配置。
- 检查 CHAR 类型的字段所定义的长度，如果超过设置，则建议转换为 VARCHAR，类型为警告，可配置。
- 检查列的类型是不是 BLOB，类型为警告，可配置。
- 检查列有没有设置 NOT NULL 属性，类型为警告，可配置。
- 检查 BLOB 类型的列是不是设置了 NOT NULL 属性，如果设置了则报错，类型为警告，不可配置。
- 检查自增列是不是无符号类型，类型为警告，可配置。如果长度小于 4，则同时建议转换为 INT 或是 BIGINT，类型为警告，可配置。
- 检查在 MySQL 5.5 及以下版本中，是不是存在多个 TIMESTAMP 类型的列，类型为错误，不可配置。

- 检查 DATETIME 类型的列, 在 MySQL 5.5 及以下版本中, 是不是设置了 default CURRENT_TIMESTAMP, 或者设置了 ON UPDATE CURRENT_TIMESTAMP, 如果是则报错, 类型为错误, 不可配置。
- 检查 TIMESTAMP 类型的列, 是不是没有设置默认值, 但设置了 ON UPDATE CURRENT_TIMESTAMP, 如果是则报错, 类型为错误, 不可配置。
- 检查不为 BLOB 类型、不是自增列又不是主键包含的列, 是不是设置了默认值, 类型为警告, 可配置。
- 检查不是 BLOB 类型的列, 是不是显式地单独设置了字符集, 类型为警告, 可配置。
- 检查自增列名, 是否为 id, 类型为警告, 可配置。
- 检查自增列的个数, 只能有一个, 类型为错误, 不可配置。

索引属性检查项

- 检查索引是否有名字, 如果没有名字则直接报错, 属于错误, 不属于警告。
- 如果索引类型为 PRIMARY, 则索引名字必须是 “PRIMARY”, 否则抛出错误。
- 检查创建的索引是否为外键, 可配置, 属于警告。
- 检查创建的索引名字是否符合前缀规则, 唯一索引需要以 “uniq” 开头, 普通索引需要以 “idx_” 开头, 参数可配置, 属于警告。
- 检查索引中列的个数, 如果超过参数设置, 就报警告。
- 检查主键所包含的列个数, 如果超过参数设定数目, 则报警告。
- 检查主键所包含的列, 如果存在列的类型不为 INT, 就报警告, 有参数可配置。
- 检查一个索引所占的空间是不是超过了 767 个字节, 如果超过则报错, 类型为错误, 不可配置。
- 检查在创建索引时, 是不是与线上目前已经存在的索引名字存在冲突, 如果是则报错, 类型为错误, 不可配置。
- 检查在创建索引或表时, 索引个数是不是超过了参数设置值, 如果是则报警告。
- 检查索引列中是否包含 BLOB 列, 类型为错误, 不可配置。
- 检查建表时, 是不是存在同名的索引, 类型为错误, 不可配置。
- 检查重复索引, 类型为错误, 不可配置。

修改表语句检查项

- 检查改表语句中，要修改的表属性类型，目前只支持修改存储引擎、修改注释、修改 AUTO INCREMENT、修改字符集，在此之外的，都不支持，类型为警告。
- 检查在一个语句块中，是不是存在对同一个表的改表语句，类型为警告，可配置。
- 检查改表类型，目前支持的有加列、加索引、删列、改表名、修改列、删索引、修改列默认值、修改表属性（第 1 点中说明了具体支持的类型），以及 CONVERT 表字符集这几种类型，其他改表类型都不支持。除此之外，都报错，类型为警告。

总结

- 针对线上 MySQL 服务器是不是 5.6 以上（包含）版本，有不同的处理策略。比如在预估影响行数时，5.6 可以直接对任何 DML 语句做 EXPLAIN 操作，而在 5.5 及以下版本中，只支持对 SELECT 语句执行 EXPLAIN 操作，并且有些 DML 语句不容易直接转换为 SELECT 语句去做 EXPLAIN，这样就导致预估行数为 0。
- 还是针对 5.6 以上版本与 5.5 版本的不同，DATETIME、TIMESTAMP 系列类型在执行时，5.5 的限制比较多，而 5.6 基本通用，所以这上面的处理可能在 5.6 及 5.5 版本上，相同语句返回的结果集是不同的（规则在 5.5 版本中，以在执行时报的错误信息为准），这与线上版本有关系。
- 规则是人定的，不能尽善尽美，也不能覆盖全部，更不可能一个规则适合所有人使用，所以通过配置参数的方式，让不同的用户做不同的选择，决定哪些才是最适合自己的。不过即使这样，我认为还是不够好，因为这些能配置的参数毕竟有限，都是内置的，想新增规则不太容易。一种方法就是通过修改源码来实现，但门槛不低，很多人因此望而却步，所以我们提供了另一种方法，即 Inception 的语法树打印，这样可以结构化地分析语句中的每一部分、每一个表达式，有了这个利器，想要增加自己的规则就不难了，并且有非常大的发挥空间，应用也不止于此，可以尽情地发挥自己的想象，更好地使用 Inception，让 Inception 最大限度地为大家服务。

49

参数变量

语法和变量

考虑到不同用户定义和使用的规范会有所不同，Inception 支持很多可配置的参数。这些配置参数都是全局参数。对于同一个服务的规则，不应该经常变化，或者说不应该出现一些业务是这样，而另一些业务是那样的规则，所以这些变量一经设置，就影响所有的审核规则。如果一家公司或一个项目确实需要有多个审核规则，那么建议配置多套 Inception 服务，在各自的配置文件中指定相应的参数值，分别进行审核。

可以通过 MySQL 客户端连接到 Inception 服务器，通过 Inception 特定语法的命令，设置或打印 Inception 变量的值。连接 Inception 的时候，只需要指定 Inception 的地址及端口即可，其他用户名密码可以不指定，因为 Inception 没有权限验证的过程。

Inception 打印变量时，不支持像原来的 MySQL 服务器一样的模糊匹配，例如 `show variables like '%name%'`。Inception 只能是精确匹配，如果找不到则返回空结果集。当然，Inception 也可以打印所有变量，语法见下表。

支持语句	意义
<code>inception get variables 'variable_name';</code>	通过 <code>variable_name</code> 指定变量名称，只显示指定的变量名的值
<code>inception get variables;</code>	显示 Inception 所有变量的值
<code>inception set variable_name=value;</code>	设置变量名为 <code>variable_name</code> 的变量的值

Inception 目前所支持的变量参数包括下面这些。

- `inception_check_insert_field`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在插入语句中, 用来控制是否指定插入列列表, 如果没有指定, 并且参数值为 ON, 则会报错。
- `inception_check_dml_where`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在审核 DML 语句时, 如果发现没有 WHERE 条件, 并且此参数设置为 ON, 就会报错, 否则被忽略。
- `inception_check_dml_limit`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能说明是在 DML 语句中, 如果使用了 LIMIT 表达式, 并且此参数设置为 ON, 就会报错。这一般用来防止 STATEMENT 语句主从复制时导致主从不一致的问题。
- `inception_check_dml_orderby`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在 DML 语句中, 如果使用了 OrderBy 表达式, 并且此参数设置为 ON, 就会报错。这一般用来防止 STATEMENT 语句主从复制时导致主从不一致的问题。
- `inception_enable_select_star`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在遇到查询语句为 “select * from”, 并且此参数设置为 ON 时, 不会报错, 否则会报错。
- `inception_enable_orderby_rand`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在语句中出现 order by rand() 时, 用来控制是否报错, 设置为 ON 表示不报错, 否则会报错。
- `inception_enable_nullable`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在创建或者新增列时, 如果列为 NULL, 用来控制是否报错, 如果设置为 ON, 表示不报错, 否则会报错。
- `inception_enable_foreign_key`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在创建表或增加索引时, 如果存在外键, 用来控制是否报错, 如果设置为 ON, 则不报错, 否则会报错。
- `inception_max_key_parts`: 参数可选范围为 1~64, 参数默认值为 5, 功能是在一个索引中, 用来控制列的最大个数, 如果超过这个数目则报错。在增加索引或新建表时, 都会生效。
- `inception_max_update_rows`: 参数可选范围为 1~MAX, 参数默认值为 10000, 功能是在一个修改语句中, 用来控制预计影响的最大行数, 如果超过这个数就报错。这个参数的获取方法是 explain, 对于有一些语句或在 MySQL 5.5 版本中获取不到相应语句时, 预计行数都会是 0, 这时这个参数就失效了。
- `inception_max_keys`: 参数可选范围为 1~1024, 参数默认值为 16, 功能在一个表中, 用来控制支持的最大索引数目, 如果超过这个数则报错, 不管在新增表, 还是新增索引时, 都有效。

- `inception_enable_not_innodb`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在新建表指定的存储引擎不是 InnoDB 时, 用来控制是否报错, 如果设置为 ON, 则不报错, 否则会报错。
- `inception_support_charset`: 参数可选范围为 MySQL 支持字符集, 参数默认值为 “utf8mb4”, 功能是表示在建表或建库时支持的字符集, 如果需要多个, 则用逗号分隔, 影响的范围是建表、设置会话字符集、修改表字符集属性等。
- `inception_check_table_comment`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表及没有设置表注释时, 用来控制是否报错, 如果设置为 ON, 则会报错。
- `inception_check_column_comment`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表或改表加列, 并且没有设置列注释时, 用来控制是否报错, 如果设置为 ON, 则会报错。
- `inception_check_primary_key`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表时, 如果没有创建主键, 用来控制是否报错, 如果设置为 ON, 就会报错。
- `inception_enable_partition_table`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在建表时, 如果创建了分区表, 用来控制是否报错, 如果设置为 ON, 不会报错, 否则会报错。
- `inception_enable_enum_set_bit`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在建表或加列时, 如果列对应的数据类型指定的是 enum、set、bit 数据类型, 用来控制是否报错, 如果设置为 ON, 则不报错, 否则会报错。
- `inception_check_index_prefix`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是用来检查新建或建表时的索引前缀, 普通索引的前缀为 “idx_”, 唯一索引的前缀为 “uniq_”, 如果设置为 ON, 并且索引前缀不符合规则, 则会报错。
- `inception_enable_autoincrement_unsigned`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在新建表时, 如果自增列不是无符号整型的数据类型, 用来控制是否报错, 如果设置为 ON, 就报错, 否则不报错。
- `inception_max_char_length`: 参数可选范围为 1~MAX, 参数默认值为 16, 功能是用来控制当 char 类型的长度大于多少时, 就提示将其转换为 VARCHAR。
- `inception_check_autoincrement_init_value`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是当建表时自增列的值指定不为 1 时, 用来控制是否报错, 如果设置为 ON, 则报错。
- `inception_check_autoincrement_datatype`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表时自增列的类型不为 int 或 bigint 时, 用来控制是否报错, 如果设置为 ON, 则会报错。

- `inception_check_timestamp_default`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表时, 如果没有为 timestamp 类型指定默认值, 用来控制是否报错, 如果设置为 ON, 则会报错。
- `inception_enable_column_charset`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在新建表或修改表加列改列时, 用来控制是否能单独指定列的字符集, 如果设置为 ON, 则表示可以设置, 不报错。
- `inception_check_autoincrement_name`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表时, 如果指定的自增列名字不为 ID, 用来控制是否报错, 如果设置为 ON, 则报错, 表示这个列可能存在业务意义, 起到提示的作用。
- `inception_merge_alter_table`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在同一个 Inception 任务中, 多个语句修改同一个表的语句出现时, 用来控制是否报错, 如果设置为 ON, 则报错, 并提示合成一个。
- `inception_check_column_default_value`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表、修改列、新增列时, 用来控制新的列属性是否要有默认值, 如果设置为 ON, 则说明必须要有默认值, 否则会报错。
- `inception_enable_blob_type`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是在建表、修改列、新增列操作时, 如果存在 BLOB 类型的列, 用来控制是否报错, 如果设置为 ON, 说明支持 BLOB 类型, 则不会报错。
- `inception_enable_identifer_keyword`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在所有审核的 SQL 语句中, 如果有标识符被写成 MySQL 的关键字, 用来控制是否报错。如果设置为 ON, 说明支持标识符为关键字, 就不会报错, 否则会报错。由于历史原因, 这里的 `identifer` 写错了, 正确写法是 `identifier`, 但 Inception 发布已久, 只能将错就错。
- `auto_commit`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是为了匹配 Python 客户端每次自动设置 `auto_commit=0` 的, 如果取消则会报错, 针对 Inception 本身没有实际意义。
- `bind_address`: 参数可选范围为 string, 参数默认值为 *。这个参数实际上就是 MySQL 数据库原来的参数, 因为 Inception 没有权限验证过程, 那么为了实现更安全的访问, 可以给 Inception 服务器的这个参数设置某些机器 (Inception 上层的应用程序) 的地址, 这样其他非法程序就是不可访问的了, 再加上 Inception 执行选项中的用户名密码, 对于后端 MySQL 就更加安全了。
- `general_log`: 参数可选范围为 ON/OFF, 参数默认值为 ON。这个参数就是原生的 MySQL 参数, 用来记录在 Inception 服务上执行过哪些语句, 定位一些问题等。

- `general_log_file`: 参数可选范围为 string, 参数默认值为 `inception.log`, 功能是设置 `general_log` 写入的文件路径。
- `inception_user`: 参数可选范围为 string, 参数默认值为 `empty`。这个用户名在配置之后, 在连接 Inception 的选项中可以不指定 `user`, 这样就可以不暴露线上数据库的用户名及密码了, 可以作为临时使用的一种方式。但这个用户现在只能用来审核, 也就是说, 即使在选项中指定 `--enable-execute`, 也不能执行, 是只能用来审核的账号。
- `inception_password`: 参数可选范围为 string, 参数默认值为 `empty`。这个参数与上面的参数是一对的, 对应的是选项中的 `password`, 设置这个参数之后, 可以在选项中不指定 `password`。
- `inception_enable_sql_statistic`: 参数可选范围为 ON/OFF, 参数默认值为 ON。用来设置是否支持在统计 Inception 执行过的语句中, 记录各种语句分别占多大比例。如果参数值为 ON, 则每次执行的情况都会在备份数据库实例中 `inception` 库的 `statistic` 表中, 以一条记录的形式存储这次操作的统计情况, 每次操作对应一条记录, 这条记录中含有的信息是各种类型的语句执行次数情况, 具体的信息请参照 52 章中 “Inception 对 SQL 执行情况的统计” 一节。
- `inception_read_only`: 参数可选范围为 ON/OFF, 参数默认值为 OFF。设置当前 Inception 服务器是否为只读, 这是为了防止一些人在具有修改权限的账号时, 通过 Inception 误修改一些数据。如果 `inception_read_only` 设置为 ON, 则即使打开了 `enable-execute`, 同时又有执行权限, 也不会去执行, 审核完成即返回。
- `inception_check_identifier`: 参数可选范围为 ON/OFF, 参数默认值为 ON, 功能是打开与关闭 Inception 对 SQL 语句中各种名字的检查。如果设置为 ON, 则发现名字中存在除数字、字母、下划线之外的字符时, 会报 Identifier “invalidname” is invalid, valid options: [a-z,A-Z,0-9,_]。
- `inception_max_primary_key_parts`: 参数可选范围为 1~64, 参数默认值为 5, 功能是在创建表时, 如果主键所包含的列个数超过这个设置的值, 则会报警告。
- `inception_enable_pk_columns_only_int`: 参数可选范围为 ON/OFF, 参数默认值为 OFF, 功能是在参数设置为 ON, 则在创建表或创建主键索引时, 会判断包含的列的类型是否只有 INT 类型的, 如果不是则报警告。

注意事项

上面已经说了, 可以用 MySQL 客户端通过命令 `inception get variables`; 查看 Inception 支持的所有参数变量, 所有以 `inception` 开头的参数, 都是专门为 Inception 加的, 而其他的则大

都是 MySQL 原生的，大部分没有任何作用的都已经去除了，有些则没有。在使用过程中可酌情处理。

而以 `inception` 开头的参数中，还有一部分是以 `inception_osc` 开头的，这十几个参数主要是用来控制 Inception 在使用 OSC 工具来执行 ALTER 表操作时使用的，这部分会在 52 章中详细介绍。

50

友好的结果集

当初在设计 Inception 时，就本着通用、接口友好的理念来实现一款使用方便、简单、功能丰富并强大的 MySQL 自动化运维软件，第 46 章中已经介绍了基本的功能，而能把这些功能体现出来，并且得到更广泛的使用，还需要一套非常友好的接口来支撑，本章所要讲述的内容——结果集，就是这个强有力支撑中最出彩的一个。

结果集结构

Inception 给用户返回的信息有如下两种。

- 异常信息：如果提交给 Inception 的基础信息存在错误，例如源信息不全、源信息有错误等，Inception 在这种情况下会直接报异常，包括错误码及错误信息。这与 MySQL 服务器返回的异常是一样的，在外面正常处理即可。
- 结果集：如果没有异常，Inception 会以结果集的方式将检查结果告诉客户端。这种方式非常友好，在它的基础上用现有的成熟接口处理起来非常方便。返回的结果集中，每一个行数据就是一条提交的 SQL 语句，Inception 内部将所有提交的语句块逐条地拆开，以结果集的方式返回。针对每一条语句，Inception 检查出的问题或审核状态，在结果集中是一目了然的。如果其中某一条 SQL 语句有任何问题，都会体现在结果集的这行信息中。



注意：如果在语句中出现语法错误，则被分析语句从这个位置开始，之后的部分都不能继续正确分析。Inception 只能认为从这个位置开始之后的所有语句是一条语句。当然这条语句对应的错误信息就是语法错误，而之前被成功分析并拆开的语句会在结果集中正常返回，并且有相应的分析结果，结果集中的最后一条记录就是语法错误的记录。

Inception 返回结果集的每一列的详细描述如下。

- **ID**: 用来表示结果集中记录序号的，也就是被审核的语句在语句块中的序号，按位置排序，计数从 1 开始。
- **stage**: 这一列显示当前语句已经进行到哪一步了，总共包括四个值，分别是 CHECKED、EXECUTED、RERUN 和 NONE。NONE 表示没有做过任何处理，有可能前面有语法错误直接提前返回了；CHECKED 表示这个语句只做过审核，而没有再进行下一步操作；EXECUTED 表示已经执行过，如果执行失败，也是用这个状态表示；RERUN 表示的是，对于影响上下文的语句，已经执行成功，但为了与 EXECUTED 区分，用 RERUN 表示，主要是在执行过程中，如果某一条语句执行失败了，则上层可能需要将没有执行的语句提取出来，再次执行，那么影响上下文的语句就是需要加上的，所以用 RERUN 来表示。Inception 目前支持两种影响上下文的语句，分别是 `set names charsetname` 语句和 `use database` 语句。
- **errlevel**: 该列目前总共有三个值，分别是 0、1、2。如果为 0，则说明当前语句审核没有任何问题；如果值为 1，则说明当前语句审核时发现有些写法不符合 Inception 定义的内置规则，属于警告；如果值为 2，则说明当前语句审核时，发现了严重错误，是无论如何都不能通过的，有这种错误时，如果想要做的是 `--enable-execute`，即使打开 `--enable-ignore-warnings`，也没有用。分等级的设计初衷是，对于规则的设定，Inception 的执行是死的。但有时候由于业务的特殊需要，不符合规则也是可以通过的，即警告是可以被忽略的，具体过程可以由 DBA 和业务开发人员商议决定。所以在实现自动化运维平台时，一个比较好的实现方式是，如果审核结果中没有 2，但有 1，开发同学可以提交成功，但经过 DBA 审核之后，如果认为不能通过，那么就可以打回修改；而对于存在 2 的情况，则可以在平台上直接被限制提交，必须是在开发同学修改过之后，才能提交成功。这种方式增大了平台的容错率，考虑到了业务环境的复杂性及特殊性。
- **stagesatus**: 该列用来描述当前语句的阶段结果，与列 stage 对应。如果是审核阶段，并且完成，则返回 Audit completed。如果是执行阶段，并且执行成功则返回 Execute Successfully，否则返回 Execute failed。如果是备份阶段，并且备份成功，则在执行描述信息后面追加 Backup successfully，否则追加 Backup failed。这一列的返回信息是为了将结果集直接输

出而设置的,如果在具体的使用过程中,为了更友好地显示,则可以在此基础上再做加工处理。

- **errormessage**: 用来表示出错的错误信息,这里包括一条语句中的所有错误信息,用换行符分隔,但有时候如果某一个错误导致不能继续分析了,比如表不存在等问题,在这种情况下,如果语句还有其他错误,就不能被审核出来了。如果当前语句没有任何错误,则这个列的值为 None。对于执行及备份操作,因为对于一条语句,这样的错误只会有一次,那么执行错误会在后面追加“execute: 具体的执行错误原因”,如果是备份出错,则在后面追加“backup: 具体的备份错误原因”。在执行时,有时候还会出现 Warnings,比如插入数据时字符串被截断等,此时会在后面输出这些 warnings,输出格式是 #1 Execute(Warning, Code errno):warning message, # 号后面的数字表示第几个警告,因为有时候执行一条语句会产生多个警告。
- **SQL:** 用来表示当前检查的是哪条 SQL 语句,这一列所存储的值就是这条 SQL 语句的文本内容。如果某一条 SQL 语句在检查时有语法错误,则这里会包括从出错语句开始到后面所有的语句。因为语法出错后,Inception 就不能再继续分析了,也就不能将后面的每条语句分开了。这一列还会有一个特别的地方,即如果当前语句是 inception show xxxx 命令,则会在这里输出命令的结果,具体使用方法会在关于“Inception 命令集”中详细讲述。
- **affected_rows**: 审核时,用来表示当前语句预计影响的行数,这个行数一般是通过 EXPLAIN 来获取的,但在 MySQL 5.5 及以下版本中,只支持 SELECT,所以对于增删改语句的影响行数,这一列的值有可能就是 0 了。在执行时,该列输出的是执行时真实影响的行数。
- **sequence**: 该列与 Inception 备份功能有关,其实就是与 `$_$Inception_backup_information$$` 表中的列 `opid_time` 一一对应,这就为自动化运维平台针对某一条语句做回滚操作找到了入口,每次执行都会产生一个序号,如果要回滚,就使用这个值从备份表中找到对应的回滚语句执行即可。
- **backup_dbname**: 该列表示的是当前语句产生的备份信息,存储在备份服务器的哪个数据库中。这是一个字符串类型的值,只针对需要备份的语句才有意义,数据库名由 IP 地址、端口、源数据库名组成,由下划线连接,而如果是需要备份的语句,则返回字符串 None,提供这个信息,主要是为了让运维平台方便,不需要自己拼接这个库名了。
- **execute_time**: 该列表示当前语句的执行时间,单位为秒,精确到小数点后两位。列类型为字符串,使用时可能需要转换成 DOUBLE 类型的值,如果只是审核而不执行,则该列返回的值为 0。
- **SQLSHA1**: 这一列用来存储当前这条语句的一个 HASH 值,用来标识这个语句是否会使用 OSC 功能,如果返回信息中有值,则表示这条语句在执行时会使用 OSC。因为在执行

前, 会有一次单独的审核操作, 此时上层已经可以拿到了这个值, 审核通过之后, 语句是不会改变的, 当然这个值也不会改变, 那么在执行时就可以使用这个值来查看 OSC 执行的进度等信息。这个 HASH 值一般的样子如下: *D0210DFF35F0BC0A7C95CD98F5BCD4D9B0CA8154, 其他具体信息, 请参考 52 章中“对 OSC 的支持”一节的详细讲述。

总结

上面的列, 是为了尽可能丰富且灵活地让上层使用, 所以列比较多。在具体使用中, 如果哪些列觉得没什么意义或用处不大, 则可以不用关心, 特别是只做一个审核页面的时候, 很多列都是不需要关心的, 只需要关心 ID、errlevel、errormessage、SQL 及 affected_rows 即可。总之一句话, 灵活应用即可。

从上面结果集的讲述中可以看出, 返回的信息非常丰富。基于此, 想要实现一个功能强大、方便简洁的自动化运维平台, 完全是够用的。没有做不到, 只有想不到的, 可以按照现实需求, 一步步增强。

其实, DBA 的工作也可以非常轻松、快乐。

51

命令集语句

Inception 是一个服务程序，启动之后，像 MySQL 服务器一样，一直监听应用程序的连接。同样地，也可以使用 MySQL 客户端来连接。Inception 服务器本身也是需要做一些运维操作的，比如最简单的设置及查看本地变量等，所以 Inception 还支持一些操作命令，可以方便地对服务器做一些操作。

Inception 命令集是 Inception 支持的一系列命令的集合，为了与原 MySQL 的命令区分，在命令前使用 Inception 这个新的关键字。

远程信息获取

这种命令，可以打印线上服务器的一些信息。目前支持所有 show 命令，只要在原来 show 命令前面加上 inception 即可。返回的结果集放在第 50 章中所述结果集的 SQL 列中，输出时具有一定的格式：前面是当前执行的 inception 命令，以冒号分隔，后面接着是执行 show 命令得到的结果集，每行以换行符分隔，一行中不同列的数据，以竖线“|”分隔。

语法如下。

```
 inception {MySQL支持的原生的show命令};
```


举例说明如下。

```
inception show tables;
inception show variables like "%read_only%";
```

这个命令可以用来方便地通过 Inception 工具查看线上的某些状态或变量等信息，返回的结果会放在结果集的 SQL 列中。比如如下命令。

```
/* --user=xxxx;--password=xxxxxxxxxxx;--host=xxxxxxxxxx;
   --enable-check;--port=3456; */
inception_magic_start;
use mysql;
inception show variables like "read%";
inception_magic_commit;
```

返回的结果集中，SQL 列的值如下所示。

```
inception show variables like "read%":
read_buffer_size | 33554432
read_only | OFF
read_rnd_buffer_size | 262144
```

因为这里是要查看远程变量，所以需要连到远程数据库服务器，执行时必须加上前面的数据源信息，并且通过 Python 等客户端写程序来执行。如果使用 MySQL 客户端直接执行，则会报错，因为客户端不支持这样的执行方式。如图 51.1 所示。

```
mysql> inception show variables like "read%" \G
***** 1. row *****
      ID: 1
    Stage: NONE
  Error_Level: 2
  Stage_Status: None
Error_Message: Must start as begin statement.
      SQL: Global environment
Affected_rows: 0
    Sequence: None
Backup_Dbnames: None
  Execute_Time: 0
      sqlsha1: None
      Command: None
1 row in set, 1 warning (0.00 sec)

mysql>
```

图 51.1

图 51.1 中的报错就是在提醒，必须要放在 inception_magic_start 及 inception_magic_commit 语句中才能执行。

显示本地全部变量

显示本地全部变量，就是查看 Inception 本身所有参数的值，可以通过 MySQL 客户端连接到 Inception，然后执行相应的命令，语法非常简单，如下所示。

```
inception get variables;
```

输出就是所有的变量，图 51.2 所示为部分变量。

```
mysql> inception get variables;
```

Variable_name	Value
autocommit	OFF
bind_address	*
character_set_system	utf8
character_sets_dir	/data/workspace/inception/debug/sql/Debug/share/charsets/
connect_timeout	10
date_format	%Y-%m-%d
datetime_format	%Y-%m-%d %H:%i:%s
explicit_defaults_for_timestamp	ON
general_log	ON
general_log_file	/data/workspace/inception_data/inception.log
inception_check_autoincrement_datatype	ON
inception_check_autoincrement_init_value	ON
inception_check_autoincrement_name	ON
inception_check_column_comment	ON
inception_check_column_default_value	ON
inception_check_dml_limit	ON
inception_check_dml_orderby	ON
inception_check_dml_where	ON
inception_check_index_prefix	ON
inception_check_insert_field	ON
inception_check_primary_key	ON
inception_check_table_comment	ON
inception_check_timestamp_default	ON
inception_ddl_support	ON
inception_enable_autoincrement_unsigned	ON

图 51.2

因为这条命令只显示 Inception 本身一些参数变量的值，所以就不需要数据源信息，可以在 MySQL 客户端直接执行这个命令。

显示本地某个变量

上面是显示了所有 Inception 支持的变量，如果想要显示指定的某一个变量，可以使用下面的语法格式。

```
inception get variables 'variable_name';
```

可以看到，和显示全部变量的语法区别很小，只需要指定要获取的变量名称即可，如图 51.3 所示。

```
mysql> inception get variables 'inception_check_primary_key';
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| inception_check_primary_key | ON    |
+-----+-----+
1 row in set (0.00 sec)
```

图 51.3

这个命令只显示 Inception 本身的一些参数变量值，所以不需要数据源信息，所以它是可以直接在 MySQL 客户端执行的，而如果放在数据源信息中执行的话，这样的语句会被忽略，不会做任何处理。

设置本地变量

Inception 不仅可以查看它的参数值，还可以通过语句动态设置某个参数的值。因为 Inception 的大部分参数是全局的，并且不是会话级别的，所以它是在设置之后马上生效的。而有些参数，包括 OSC 的参数是会话的，设置之后，只影响当前执行的连接。语法如下。

```
inception set [session] variables_name=value;
```

这个命令只设置了 Inception 本身的一些参数变量值，所以不需要数据源信息，是可以直接在 MySQL 客户端执行的。而如果放在数据源信息中执行的话，这样的语句会被忽略，不会做任何处理。如图 51.4 所示。


```
mysql> inception set session inception_osc_max_lag=0;
Query OK, 0 rows affected (0.00 sec)
```

图 51.4

如语法中所示，如果指定了 session，则说明是要修改当前连接的对应参数变量，不会影响其他连接。这一般是在向 Inception 批量提交任务之前，先将这些参数设置好，然后再使用同一个连接来提交，保证修改生效。

显示 OSC 执行进度

下面这条语句的作用是,当某一个 ALTER TABLE 语句正在通过 Inception 使用 OSC 执行时,可以通过这条语句来查询执行的进度信息,语法如下。

 `inception get osc_percent '当前执行的SQL语句以及一些基本信息生成的SHA1哈希值';`

具体的细节会在 52 章中讲述,这里只看一下图例,如图 51.5 所示。

```
mysql> inception get osc_percent '*98A11AC683C0D121568A51CA33A3A94674326630'\G
***** 1. ROW *****
  DBNAME: sbtest
  TABLENAME: sbtest1
  SQLSHA1: *98A11AC683C0D121568A51CA33A3A94674326630
  PERCENT: 7
REMAINTIME: 01:57
INFOMATION: No slaves found. See --recursion-method if host ##### has slaves.
Not checking slave lag because no slaves were found and --check-slave-lag was not specified.
Operation, tries, wait:
  copy_rows, 10, 0.25
  create_triggers, 10, 1
  drop_triggers, 10, 1
  swap_tables, 10, 1
  update_foreign_keys, 10, 1
No foreign keys reference `sbtest`.`sbtest1`; ignoring --alter-foreign-keys-method.
Altering `sbtest`.`sbtest1`...
Creating new table...
Created new table sbtest.____sbtest1_new OK.
Altering new table...
Altered `sbtest`.`____sbtest1_new` OK.
2015-09-16T17:48:48 Creating triggers...
2015-09-16T17:48:48 Created triggers OK.
2015-09-16T17:48:48 Copying approximately 491522 rows...

1 row in set (0.00 sec)
```

图 51.5

上面是正在做的操作,而如果语句块中有多个修改表的操作,则在输出的多条记录中,会看到已经执行完成的进度信息,如图 51.6 所示。

```
mysql> inception get osc_percent '*98A11AC683C0D121568A51CA33A3A94674326630'\G
***** 1. ROW *****
DBNAME: sbtest
TABLENAME: sbtest1
SQLSHA1: *98A11AC683C0D121568A51CA33A3A94674326630
PERCENT: 100
REMAINTIME: 00:00
INFOMATION: No slaves found. See --recursion-method if host ##### has slaves.
Not checking slave lag because no slaves were found and --check-slave-lag was not specified.
Operation, tries, wait:
  copy_rows, 10, 0.25
  create_triggers, 10, 1
  drop_triggers, 10, 1
  swap_tables, 10, 1
  update_foreign_keys, 10, 1
No foreign keys reference `sbtest`.`sbtest1`; ignoring --alter-foreign-keys-method.
Altering `sbtest`.`sbtest1`...
Creating new table...
Created new table sbtest.___sbtest1_new OK.
Altering new table...
Altered `sbtest`.`___sbtest1_new` OK.
2015-09-16T17:48:48 Creating triggers...
2015-09-16T17:48:48 Created triggers OK.
2015-09-16T17:48:48 Copying approximately 491522 rows...
2015-09-16T17:51:15 Copied rows OK.
2015-09-16T17:51:15 Swapping tables...
2015-09-16T17:51:15 Swapped original and new tables OK.
2015-09-16T17:51:15 Dropping old table...
2015-09-16T17:51:15 Dropped old table `sbtest`.`_sbtest1_old` OK.
2015-09-16T17:51:15 Dropping triggers.
2015-09-16T17:51:15 Dropped triggers OK.
# Event Count
# =====
# INSERT 1243
Successfully altered `sbtest`.`sbtest1`.

1 row in set (0.00 sec)
```

图 51.6

查看当前 processlist

Inception 还可以查看当前正在执行的所有线程信息，语法如下。

```
inception get processlist;
```

当前命令执行之后，返回的结果信息如图 51.7 所示，每一列的介绍如下。

- id: 一个简单的计数。
- dest_user: 表示当前执行语句访问后端数据库时所用的用户名。
- dest_host: 表示当前执行语句要访问的后端数据库地址。
- dest_port: 表示当前执行语句要访问的后端数据库的端口。

- `from_host`: 表示当前执行语句是从哪个机器上发起的。一般情况下是自动化运维平台程序部署的机器地址, 如果有多套, 则会出现不同的 `from_host`。
- `command`: 表示当前执行的是什么操作, 包括 CHECK (简单审核)、EXECUTE (执行)、SPLIT (拆分)、PRINT (打印计划树) 和 LOCAL (本地命令)。
- `state`: 表示在当前命令下, 执行的状态是什么, 状态包括 INIT (初始阶段)、CHECKING (正在审核)、EXECUTING (正在执行)、DEINIT (退出) 和 BACKUP (正在备份)。
- `time`: 表示当前语句执行所用的时间。
- `info`: 显示当前正在执行的语句。
- `current_Execute`: 现在已经清楚 Inception 的运作过程了, 先审核, 然后执行, 最后做备份。但是执行和备份都是在审核完成后, 或者语句 `inception_magic_commit` 中做的。这样, `PROCESSLIST` 中的列 `info` 只会显示为语句 `inception_magic_commit`, 而具体执行哪一条语句, 就不知道了。所以 Inception 为了更好更明确地提供服务, 用这一列来展示当前正在执行的列, 可以更好地把握总的执行进度。

```
mysql> inception get processlist\G
***** 1. row *****
      Id: 1047126
    Dest_User: #####
    Dest_Host: #####
    Dest_Port: 3308
    From_Host: localhost
      Command: EXECUTE
        STATE: BACKUP
        Time: 1615832
        Info: INCEPTION_MAGIC_COMMIT
    Current_Execute: delete from ##### where createtime
>= '2016-10-31 23:55:00' and createtime < '2016-11-
***** 2. row *****
      Id: 2244948
    Dest_User:
    Dest_Host:
    Dest_Port: 0
    From_Host: localhost
      Command: LOCAL
        STATE: CHECKING
        Time: 0
        Info: inception get processlist
    Current_Execute: NULL
2 rows in set (0.00 sec)
```

图 51.7

52

Inception 的彩蛋

在实现 Inception 的过程中，我们顺手实现了一些有意思的小功能。这些功能看似简单，但如果真的想自己动手编程去实现它，却是不太容易的。然而，在依赖 Inception 的整个框架下，借助现有的一些程序功能模块，再加入一些自己的逻辑，就有事半功倍的奇效。

对 OSC 的支持

对 MySQL 的 DDL 和 DML 语句进行审核之后，自然而然就是要到线上去操作了。在 Inception 中把这部分操作也集成了进来，提供连接到线上数据库服务器执行相应 SQL 语句接口的功能。如果配合合适的界面程序或对接口进行服务化的封装，就很容易通过 Inception 实现对线上数据库 SQL 的审核、执行及后续备份的一体化操作了。

在执行过程中，有一件非常值得注意的事情就是，对非常耗时的 DDL 的处理。为了保证对线上数据库影响最小，在 Inception 里面集成了 Percona ToolKit 工具 `pt-online-schema-change`。这样对于大表的修改操作，就不需要跳过 Inception 去手动执行了，再次给线上操作带来了非常大的方便。接下来，说明一下对 OSC 支持的一些细节。

可选的 OSC 参数

为了更友好地实现对 OSC 的集成，增加了如下的一些参数。

参数名称	作用域	默认值	说明
inception_osc_bin_dir	GLOBAL	无	用于指定 pt-online-schema-change 脚本的位置, 不可修改, 在配置文件中设置
inception_osc_check_interval	SESSION	5 秒	对应参数--check-interval, 意义是 Sleep time between checks for --max-lag
inception_osc_chunk_size	SESSION	1000	对应参数--chunk-size
inception_osc_chunk_size_limit	SESSION	4	对应参数--chunk-size-limit
inception_osc_chunk_time	SESSION	1	对应参数--chunk-time
inception_osc_critical_thread_connected	SESSION	1000	对应参数--critical-load 中的 thread_connected 部分
inception_osc_critical_thread_running	SESSION	80	对应参数--critical-load 中的 thread_running 部分
inception_osc_drop_new_table	SESSION	1	对应参数--[no]drop-new-table
inception_osc_drop_old_table	SESSION	1	对应参数--[no]drop-old-table
inception_osc_check_replication_filters	SESSION	1	对应参数--[no]check-replication-filters
inception_osc_check_alter	SESSION	1	对应参数--[no]check-alter
inception_osc_max_lag	SESSION	3	对应参数--max-lag
inception_osc_max_thread_connected	SESSION	1000	对应参数--max-load 中的 thread_connected 部分
inception_osc_max_thread_running	SESSION	80	对应参数--max-load 中的 thread_running 部分
inception_osc_recursion_method	SESSION	processlist	对应 OSC 参数 recursion_method, 具体意义可以参考 OSC 官方手册
inception_osc_alter_foreign_keys_method	SESSION	none	对应 OSC 参数 alter-foreign-keys-method, 具体意义可以参考 OSC 官方手册

续表

参数名称	作用域	默认值	说明
inception_osc_min_table_size	SESSION	16	这个参数实际上是一个 OSC 开关, 如果设置为 0, 则全部 ALTER 语句都使用 OSC 方式, 如果设置为非 0, 则当这个表占用空间大小大于这个值时才使用 OSC 方式。单位为 M, 这个表大小的计算方式是通过语句 <code>select (DATA_LENGTH + INDEX_LENGTH)/1024/1024 from information_schema.tables where table_schema = 'dbname' and table_name = 'tablename'</code> 来实现的
inception_osc_on	GLOBAL	1	一个全局的 OSC 开关, 默认是打开的, 如果想要关闭则设置为 OFF, 这样就会直接修改
inception_osc_print_sql	GLOBAL	1	对应参数为--print
inception_osc_print_none	GLOBAL	1	用来设置在 Inception 返回结果集中, 对于原来 OSC 在执行过程的标准输出信息是不是要打印到结果集对应的错误信息列中, 如果设置为 1, 就不打印, 如果设置为 0, 就打印。而如果出现了错误, 则都会打印

参数名称、作用域、默认值及意义在上面都已经列出来了。针对全局的参数, 比如 `inception_osc_on`, 都是用来控制所有的 OSC 行为的, 因为大部分参数是 SESSION 级的, 所以在提交具有改表语句的任务时, 首先需要获取 Inception 连接, 然后通过这个连接线程, 修改自己的这些参数值, 语句为 `inception set session session_variable_name=value`, 然后才能开始执行, 这样才能保证改表 SESSION 的这些参数生效。

查看 OSC 的执行进度

对于 ALTER 比较大的表, 因为所用时间比较长, 所以在修改时可能需要关注一下进度。而 OSC 工具本身在执行时是打印进度信息的, 所以 Inception 完全可以提供这方面的信息。出

于更友好的实现方式, 在 Inception 中新加入一条语句, 可以查询当前执行的语句进度, 语句语法如下。

 `inception get osc_percent '当前执行的SQL语句以及一些基本信息生成的SHA1哈希值'`

通过这个语句, 可以查看进度信息, 这个返回的结果集包括如下五列。

- **TABLENAME:** 当前被修改的表名。
- **DBNAME:** 当前被修改表所在的库名。
- **SQLSHA1:** 当前要查询语句的 SHA1 字符串。
- **PERCENT:** 当前修改已经完成的百分比, 这个值是 0 到 100 的值。
- **REMAINTIME:** 当前修改语句还需要多久才能完成, 如 03:55 表示还需要 3 分 55 秒, 01:33:44 表示还需要 1 小时 33 分 44 秒。
- **INFOMATION:** 显示当前 OSC 执行时的状态信息, 内容为 OSC 当前所有的输出信息, 不包括百分比信息, 百分比还是由上面的列来显示, 这方便在使用时随时查看执行到了哪一步, 可以更加清楚地了解执行进度。

图 52.1 所示是在 ALTER 的时候, 查到的正在做的信息。

```
mysql> inception get osc_percent '*98A11AC683C0D121568A51CA33A3A94674326630'\G
***** 1. ROW *****
  DBNAME: sbtest
TABLENAME: sbtest1
  SQLSHA1: *98A11AC683C0D121568A51CA33A3A94674326630
  PERCENT: 7
REMAINTIME: 01:57
INFORMATION: No slaves found. See --recursion-method if host ##### has slaves.
Not checking slave lag because no slaves were found and --check-slave-lag was not specified.
Operation, tries, wait:
  copy_rows, 10, 0.25
  create_triggers, 10, 1
  drop_triggers, 10, 1
  swap_tables, 10, 1
  update_foreign_keys, 10, 1
No foreign keys reference `sbtest`.`sbtest1`; ignoring --alter-foreign-keys-method.
Altering `sbtest`.`sbtest1`...
Creating new table...
Created new table sbtest.____sbtest1_new OK.
Altering new table...
Altered `sbtest`.`____sbtest1_new` OK.
2015-09-16T17:48:48 Creating triggers...
2015-09-16T17:48:48 Created triggers OK.
2015-09-16T17:48:48 Copying approximately 491522 rows...

1 row in set (0.00 sec)
```

图 52.1

图 52.1 所示的是正在做的信息，而如果语句块中有多个修改表的操作，则前面的会看到执行完成的进度信息，如图 52.2 所示。

```
mysql> inception get osc_percent '*98A11AC683C0D121568A51CA33A3A94674326630'\G
***** 1. row *****
  DBNAME: sbtest
  TABLENAME: sbtest1
  SQLSHA1: *98A11AC683C0D121568A51CA33A3A94674326630
  PERCENT: 100
  REMAINTIME: 00:00
  INFOMATION: No slaves found. See --recursion-method if host ##### has slaves.
  Not checking slave lag because no slaves were found and --check-slave-lag was not specified.
  Operation, tries, wait:
    copy_rows, 10, 0.25
    create_triggers, 10, 1
    drop_triggers, 10, 1
    swap_tables, 10, 1
    update_foreign_keys, 10, 1
  No foreign keys reference `sbtest`.`sbtest1`; ignoring --alter-foreign-keys-method.
  Altering `sbtest`.`sbtest1`...
  Creating new table...
  Created new table sbtest.____sbtest1_new OK.
  Altering new table...
  Altered `sbtest`.`____sbtest1_new` OK.
  2015-09-16T17:48:48 Creating triggers...
  2015-09-16T17:48:48 Created triggers OK.
  2015-09-16T17:48:48 Copying approximately 491522 rows...
  2015-09-16T17:51:15 Copied rows OK.
  2015-09-16T17:51:15 Swapping tables...
  2015-09-16T17:51:15 Swapped original and new tables OK.
  2015-09-16T17:51:15 Dropping old table...
  2015-09-16T17:51:15 Dropped old table `sbtest`.`_sbtest1_old` OK.
  2015-09-16T17:51:15 Dropping triggers.
  2015-09-16T17:51:15 Dropped triggers OK.
  # Event Count
  # =====
  # INSERT 1243
  Successfully altered `sbtest`.`sbtest1`.

1 row in set (0.00 sec)
```

图 52.2

具体在上层应用使用时，可以选择性地使用所需要的列。

至于上面提到的 SHA1 值是如何获得的问题，我们知道，在应用提交语句时，都会在审核通过后才能提交到流程管理数据库中，那么此时这条语句就不会被修改了，而在审核时返回的结果集中，有一列名叫 sqlsha1，它的作用就是用来表示当前语句要通过 OSC 的方式来执行，这个值与同一行中的“SQL”列是一一对应的，并且是唯一的，可以在后面获取任务进度时使用。

下面来看一下 ALTER 语句在满足使用 OSC 的情况下，审核时返回的结果集信息。

```
['ID', 'stage', 'errlevel', 'stagesstatus', 'errormessage', 'SQL', 'Affected_rows',
 'sequence', 'backup_dbname', 'execute_time', 'sqlsha1']
1 | CHECKED | 0 | Audit completed | None | use sbtest | 0 | '0_0_0' | None | 0 |
2 | CHECKED | 0 | Audit completed | None | alter table sbtest1 add c2 int not null
  default 'a' comment 'for test' | 449234 | '0_0_1' | 127_0_0_1_3306_sbtest | 0 |
*F270A6902BB3A0E2DE042A60D79F55418C8D1C00
```

其中，“*98A11AC683C0D121568A51CA33A3A94674326630”就是上面 ALTER 语句对应的 SQLSHA1 值。

当进入执行阶段后，在执行改表语句时，Inception 会一直监听 OSC 的输出信息并从中找到关于进度的信息。Inception 在拿到进度信息后，会根据当前语句的 SHA1 值更新 Inception 内存中缓存对应的 ALTER TABLE 进度信息，内部设置的是，Inception 会让 OSC 每百分之一返回一次进度信息，每返回一次 Inception 就会更新一次当前语句对应的进度信息。一个小的细节是，OSC 不会返回 100%，最大 99%，此处 Inception 的处理是，当检查到有 successfully altered 的信息之后，就将进度信息改为 100%，剩余时间为 00:00，而 99% 到 100% 之间做的事情包括清除环境的操作，所以时间可能比之前的 1% 要长。

进度信息缓存有生命周期，在整个语句块执行完成后，相应的缓存信息就会被清除出去，之后再查询进度就查不到了。查不到的话，当前线程的阻塞也就返回了，说明已经完成，或者执行失败了。

中止 OSC 的执行

在执行 OSC 的过程中，有可能遇到的问题是，执行了一部分，突然发现对线上造成了 MDL 等待的现象。这种影响对一些业务是不可接受的，因为很多语句此时就不能执行当前表上的任何操作了，必须要等 OSC 的一些辅助操作（建立/删除触发器）完成之后才可以，而 OSC 的这些操作又是在等待线上一些慢查询语句执行完成之后才能继续执行下去。这种情况下，一般的处理方式是，先退出 OSC 执行，压力小的时候多试几次，才可以继续执行下去，此时最需要 DBA 操作的就是取消当前这个 OSC 的执行，所以 Inception 支持 OSC 执行的中止功能。

取消方式与查询执行进度是一样的，还是通过一个新的 Inception 命令来实现，新的命令如下。

```
inception stop alter '当前执行的SQL语句以及一些基本信息生成的SHA1哈希值'
```


这里用到的还是那个 SQLSHA1，当 OSC 开始执行时，这个 SQLSHA1 对应的缓存对象会被加入到 OSC 缓存中，此时就可以查到执行进度了（当然一开始还是 0%）。同时，这时也就可以取消这条语句的执行了，但取消的前提是，OSC 执行的进程已经创建，另外，执行进度不到 100% 才可以取消，否则就会报错误“Can not find OSC executing task”。如果可以 kill 的话，Inception 就会主动 kill 来执行 OSC 的进程，这样轻而易举地就取消 OSC 的执行了。

但是在取消之后，还有几点需要注意，如下。

1. 在多个 ALTER 语句一起执行的情况下，如果取消某一个，那么整个执行过程就都会中止，同时被取消的语句在返回的结果集中是未执行状态。
2. 在取消语句的错误描述信息中，报错为“Execute has been abort in percent: 已执行比例, remain time: 剩余时间”。
3. 在取消之后，当前语句之后的所有语句都不会执行，当然状态为未执行。
4. 被取消语句，在取消之后，结果集 stagestatus 列的信息会设置为“Execute Aborted”。

查看所有 OSC 执行信息

语句如下。

 `inception get osc processlist;`

这条语句的功能是打印所有当前正在使用 OSC 执行的操作，如果在同一个 inception 请求中有多条 ALTER 语句，那么显示出来的就有可能存在执行进度为 100% 的语句，通过这条语句，可以轻松查看当前每一个 OSC 的执行进度，如果进度一直不前进（比如存在从库复制延迟，导致 OSC 长时间 waiting），则在这里可以看到具体信息。

实际上看到的信息和上面 `inception get osc_percent` '当前执行的 SQL 语句以及一些基本信息生成的 SHA1 哈希值' 返回的结果是一样的，只是这个返回了所有正在执行的 ALTER TABLE 语句信息。



注意：OSC 全局参数最好不要频繁修改，因为针对某一条语句的 SHA1 值是在审核阶段生成的，如果在审核的时候因为参数的原因而判断不通过 OSC 改表，则不会生成 SHA1，那么在执行时候，这条语句的进度就不能被查询到，从而影响执行过程的体验。当然这个影响也不大，因为如果执行成功了，并且参数 `inception_osc_print_none` 为 OFF，则会看到 OSC 输出的所有信息。

Inception 对 SQL 执行情况的统计

在使用了 Inception 之后，数据变更已经全面进入全自动化模式。有了这个利器之后，一些以往想做而做不成的事情，现在就可以做了，比如对 SQL 执行情况的统计，修改表的语句究竟占多大比例，或者数据变更占多大比例，使用 Inception 就可以轻而易举解决这件事情了，并且非常准确。

在 Inception 里面实现这个功能非常方便，方便到只需要打开一个参数就行了。SQL 执行统计功能涉及的参数为 `inception_enable_sql_statistic`，这个参数在第 49 章中已经介绍过。

另外, 开启统计功能。除了上面的参数之外, 还需要开启操作备份功能, 因为这些统计数据需要存储到备份数据库中。存储在名为 `inception` 的数据库中, 这个数据库主要是用来存储 Inception 的一些统计信息的, 现在只有一个表 `statistic`, 这个表存储的就是 SQL 执行数目的统计数据, 以后有可能还会做一些扩展从而新生成一些表。

`statistic` 表的结构如图 52.3 所示。

```
mysql> desc statistic;
```

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
optime	timestamp	NO		CURRENT_TIMESTAMP	
usedb	int(11)	NO		0	
deleting	int(11)	NO		0	
inserting	int(11)	NO		0	
updating	int(11)	NO		0	
selecting	int(11)	NO		0	
altertable	int(11)	NO		0	
renaming	int(11)	NO		0	
createindex	int(11)	NO		0	
dropindex	int(11)	NO		0	
addcolumn	int(11)	NO		0	
dropcolumn	int(11)	NO		0	
changecolumn	int(11)	NO		0	
alteroption	int(11)	NO		0	
alterconvert	int(11)	NO		0	
createtable	int(11)	NO		0	
droptable	int(11)	NO	PRI	0	
createdb	int(11)	NO	PRI	0	
truncating	int(11)	NO	PRI	0	

20 rows in set (0.00 sec)

图 52.3

从每一列的名字就可以看到, 其值对应的操作是什么。第一列是一个自增列, 第二列 `optime` 是操作时间, 这一列表示任务执行时间点, 可以用来统计在某一段时间内的某一个操作所占的比例大小。

下面解释一下有些不太明确的列的意义。

- **deleting:** 包括普通的删除操作及多表删除操作。
- **inserting:** 包括单行插入、多行插入及查询插入。
- **updating:** 包括普通单表更新及多表的更新。
- **renaming:** ALTER table 语句中的 rename 操作。

- **createindex**: ALTER table 语句中的 add index 操作。
- **dropindex**: ALTER table 语句中的 drop index 操作。
- **alteroption**: ALTER table 语句中修改表属性的操作, 比如存储引擎、自增值及字符集中的操作。
- **alterconvert**: ALTER table 语句中修改表字符集的操作。

对于 ALTER TABLE 操作, 因为这个操作包含很多的子操作, 比如 rename、drop index、engine innodb 等操作, 所以对于列 altertable, 其值为 renaming、createindex、dropindex、addcolumn、dropcolumn、changecolumn、alteroption 和 alterconvert 的和, 而后面的是对 ALTER TABLE 语句的细分操作统计。

在实际运用中, 这个表该怎么使用, 就要看不同用户的需求了。



注意: 因为 Inception 的审核、执行及备份是分阶段的, 只有前面的阶段执行成功之后, 才能进入下一个步骤, 具体顺序是: 先审核, 再执行, 然后做统计信息的存储, 最后是备份的存储。可以看出, 当前面执行、审核有问题从而导致提前返回时, 统计是不会被更新的, 只有成功执行了, 才会被记录下来。而如果备份出错导致提前返回, 则统计信息会被成功存储。

打印语法树

在实际工作中, 经常会有对 SQL 语句进行格式化、结构化的需求。但通常情况下, 对 SQL 语句做分析, 是一个门槛较高的工作, 所以业界现在很少有工具可以满足我们这样的需求。

在推出 Inception 之后, 这种工作及使用模式使得在 Inception 里面做这件事是非常合适的, 它具备了先天的条件。可以通过将需要结构化的 SQL 语句提交到 Inception 中, Inception 通过分析翻译转换为应用程序可以识别的类型, 然后返回给客户端。此时 DBA 或运维程序就可以根据其返回的结构化 SQL 语句, 更好地理解 SQL 的结构和执行逻辑, 也可以据此自定义一些自己的规则, 来让 Inception 满足更多人的需求。

通过这样的转换, Inception 很好地将分析 SQL 语句这样具有高门槛的事情, 巧妙地转换为一个结构化的、程序可轻易解析的工作, 这样就给我们自动化程序提供了很多机会。除了 Inception 内置的、已经定义好的规则之外, DBA 也可以捕捉并分析结构化的信息, 进一步做更具个性化的审核, 这个工作对 Inception 的可扩展能力做了很大贡献。


结果集信息

使用方法和--enable-check 等是一样的，并且在前面介绍关于 Inception 支持的选项中已经讲过了，可以通过设置选项--enable-query-print 来启用打印语法树的功能。这是一个独立的功能，只负责进行语法树分析，所以它与其他 enable 开头的选项是互斥的，不能同时设置。开启这个选项之后，再连接 Inception，执行返回的结果集是一个与拆分、审核及执行都不同的结果集，该结果集所包括的列如下。

- ID：用来表示当前语句的一个序列值。
- STATEMENT：用来存储当前被分析的 SQL 语句。
- ERRLEVEL：用来存储当打印遇到问题时错误的级别，与审核结果集中的 ERRLEVEL 意义相同。
- QUERY_TREE：当前语句的分析结果，格式为 JSON 字符串。
- ERRMSG：与上面的 ERRLEVEL 对应，当出错时，这里存储分析过程中的所有错误信息，与审核结果集中的同名列意义相同。

举例说明

举例说明才是最有说服力的。针对如下的 SQL 语句举例说明。

```
 insert into t (sno,name)
      select sno, name from t alias_t
      where sno=(
          select sno+1 from my
          where
          name like "%zhufeng%" and
          sno > '10010' and
          name=alias_t.name
      )
      order by name
      limit 100, 10;
```

执行时，提交给 Inception 的语句如图 52.4 所示。

执行结果如图 52.5 所示。



注意：使用到的 print.py 其实与介绍使用方法时的脚本没什么两样，只是显示的结果集不同而已。


```

zhufeng@zhufengmac:/data/workspace/inception_data$ cat sqlprint
/* --user=root; --password=zhufeng; --host=127.0.0.1; --port=3306; --enable-query-print; */
inception_magic_start;
use local;
insert into t (sno,name)
    select sno, name from t alias_t
    where sno=(
        select sno+1 from my
        where
            name like "%zhufeng%" and
            sno > '10010' and
            name=alias_t.name
    )
    order by name
    limit 100, 10;
inception_magic_commit;

```

图 52.4

```

zhufeng@zhufengmac:/data/workspace/inception_data$ ./print.py
['ID', 'statement', 'errlevel', 'query_tree', 'errmsg']
1 | insert into t (sno,name)
    select sno, name from t alias_t
    where sno=(
        select sno+1 from my
        where
            name like "%zhufeng%" and
            sno > '10010' and
            name=alias_t.name
    )
    order by name
    limit 100, 10 | 0 | {"command":"insert","table_object":{"db":"local","table":"t"}
,"fields":[{"type":"FIELD_ITEM","db":"local","table":"t","field":"sno"}, {"type":"FIELD_IT
EM","db":"local","table":"t","field":"name"}], "select_insert_values":{"select_list":[{"ty
pe":"FIELD_ITEM","db":"local","table":"t","field":"sno"}, {"type":"FIELD_ITEM","db":"local
","table":"t","field":"name"}], "table-ref":[{"type":"physical","db":"local","table":"t"}]
,"where":[{"type":"FUNC_ITEM","func":"=", "args":[{"type":"FIELD_ITEM","db":"local","tabl
e":"t","field":"sno"}, {"type":"SUBSELECT_ITEM","engine":"single_select","subselect":{"sel
ect_list":[{"type":"FUNC_ITEM","func":"OTHERS","name":"+","args":[{"type":"FIELD_ITEM","d
b":"local","table":"my","field":"sno"}, {"type":"INT_ITEM","value":"1"}]}], "table-ref":[{"
type":"physical","db":"local","table":"my"}], "where":[{"type":"COND_ITEM","func":"AND","a
rgs":[{"type":"FUNC_ITEM","func":"LIKE","args":[{"type":"FIELD_ITEM","db":"local","table
":"my","field":"name"}, {"type":"STRING_ITEM","value":"%zhufeng%"}]}, {"type":"FUNC_ITEM","f
unc":">","args":[{"type":"FIELD_ITEM","db":"local","table":"my","field":"sno"}, {"type":"
STRING_ITEM","value":"10010"}]}], {"type":"FUNC_ITEM","func":"=", "args":[{"type":"FIELD_IT
EM","db":"local","table":"my","field":"name"}, {"type":"FIELD_ITEM","db":"local","table":"
t","field":"name"}]}]}]}], "OrderBy":[{"type":"FIELD_ITEM","db":"local","table":"t","fi
eld":"name"}], "limit":{"limit":[{"type":"INT_ITEM","value":"10"}], "limit_offset":[{"type
":"INT_ITEM","value":"100"}]}]} | None

```

图 52.5

对应 JSON 可视化的 query_tree 如下。

```
{
  "command": "insert",
  "table_object": {
    "db": "local",
    "table": "t"
  },
  "fields": [
    {
      "type": "FIELD_ITEM",
      "db": "local",
      "table": "t",
      "field": "sno"
    },
    {
      "type": "FIELD_ITEM",
      "db": "local",
      "table": "t",
      "field": "name"
    }
  ],
  "select_insert_values": {
    "select_list": [
      {
        "type": "FIELD_ITEM",
        "db": "local",
        "table": "t",
        "field": "sno"
      },
      {
        "type": "FIELD_ITEM",
        "db": "local",
        "table": "t",
        "field": "name"
      }
    ]
  },
  "table_ref": [
    {
      "db": "local",
      "table": "t"
    }
  ]
}
```

```

    }
  ],
  "where": [
    {
      "type": "FUNC_ITEM",
      "func": "=",
      "args": [
        {
          "type": "FIELD_ITEM",
          "db": "local",
          "table": "t",
          "field": "sno"
        },
        {
          "type": "SUBSELECT_ITEM",
          "engine": "single_select",
          "subselect": {
            "select_list": [
              {
                "type": "FUNC_ITEM",
                "func": "OTHERS",
                "name": "+",
                "args": [
                  {
                    "type": "FIELD_ITEM",
                    "db": "local",
                    "table": "my",
                    "field": "sno"
                  },
                  {
                    "type": "INT_ITEM",
                    "value": "1"
                  }
                ]
              }
            ]
          }
        }
      ],
      "table_ref": [
        {
          "db": "local",
          "table": "my"
        }
      ]
    }
  ]
}

```

```

    ],
    "where": [
        {
            "type": "COND_ITEM",
            "func": "AND",
            "args": [
                {
                    "type": "FUNC_ITEM",
                    "func": "LIKE",
                    "args": [
                        {
                            "type": "FIELD_ITEM",
                            "db": "local",
                            "table": "my",
                            "field": "name"
                        },
                        {
                            "type": "STRING_ITEM",
                            "value": "%zhufeng%"
                        }
                    ]
                }
            ],
            "type": "COND_ITEM",
            "func": ">",
            "args": [
                {
                    "type": "FIELD_ITEM",
                    "db": "local",
                    "table": "my",
                    "field": "sno"
                },
                {
                    "type": "STRING_ITEM",
                    "value": "10010"
                }
            ]
        },
        {
            "type": "FUNC_ITEM",
            "func": "=",

```

```

        "args": [
            {
                "type": "FIELD_ITEM",
                "db": "local",
                "table": "my",
                "field": "name"
            },
            {
                "type": "FIELD_ITEM",
                "db": "local",
                "table": "t",
                "field": "name"
            }
        ]
    },
    ],
    "OrderBy": [
        {
            "type": "FIELD_ITEM",
            "db": "local",
            "table": "t",
            "field": "name"
        }
    ],
    "limit": {
        "limit": [
            {
                "type": "INT_ITEM",
                "value": "10"
            }
        ],
        "limit_offset": [
            {
                "type": "INT_ITEM",

```

```

        "value": "100"
    }
]
}
}
}

```

以上的 SQL 语句实际上没有任何意义, 这里只是为了尽可能好地将每一类型的表达式打印出来而随意构造的。

可以看到, 这个 JSON 串很长, 不过在结构化之后, 整个语句就非常清楚了。是什么语句类型, 用到什么表, 什么列, 有没有 ORDER BY 等, 都非常明确, 分析语句再也不是难事了, 使用程序对这个结构化的语句做分析, 应该就很容易了, 并且是非常准确的。

标签定义

不过这里还是要简单讲一下语法树 JSON 串中的一些标签, 如下。

- **command**: 每条语句都是以 **command** 开头的, 它表示是什么语句类型, 现在支持的有 **insert**、**update**、**delete**、**select** 这四种类型。
- **table_object**: 表示当前语句对哪个表做的操作, 它只针对插入、删除操作, 比如是插入哪个表, 删除哪个表。而更新操作在语法树中不太好确认哪些表被改了, 所以这里没有明确拿出来, 而是可以通过从更新列的信息中取到表信息, 这就是被更新的表。这是一个字典, 里面包括的是一个或多个表信息, 并且已经对应到其对应的数据库。
- **fields**: 表示插入语句中指定的要插入的字段列表, 如果没有指定, 则没有这个信息。它是一个数组, 包括了语句中所指定的所有列信息, 每一列都有完整的信息, 包括数据库、表及列名。因为这是一个表达式, 所以其表达式类型为 **FIELD_ITEM**, 后面会专门列出所有支持的表达式信息。
- **select_insert_values**: 这表示的是查询插入的查询部分, 它是一个字典, 里面包括了这个查询语句的所有元素, 包括查询列、查询涉及的表、WHERE 及 ORDER BY 等信息。
- **select_list**: 表示当前查询语句 (或子查询) 要查询的表达式信息, 这里可以是列, 也可以是其他计算出来的值, 例子中就有 **select sno+1...** 这样的查询。
- **table_ref**: 表示当前语句上下文中使用到的表信息, 是一个数组, 包括了所有的表, 这里所谓的上下文, 可以简单理解为, 在一个子查询的可见范围内的所有表达式, 都是属于同一个层次的。如果一个列在当前上下文中找不到, 那么可能就需要到上一层 (父亲层) 的上下文中找, 如果找到了, 就算作相关子查询, 对于这种一条语句中有不同层级的情况, 就存在不同的上下文。在例子中也有相关反映。

- `where`: 表示的是查询表达式（包括查询、删除、更新及查询等）的语法树，因为 `WHERE` 语句其实就是一个表达式，只是有可能是多个表达式的逻辑运算而已。
- `orderBy`: 表示查询时使用到的排序列，是一个数组。
- `limit`: 表示在查询时使用到的 `LIMIT` 信息，因为 `LIMIT` 是一个复合信息，包括了限制行数及开始位置等，所以会有 `limit` 及 `limit_offset`，而 `limit_offset` 有可能没有，只有限制行数。
- `groupBy`: 表示查询时使用到的分组列，是一个数组。
- `having`: 表示查询时，使用到的 `having` 表达式。
- `subselect`: 如果使用到子查询，则它就用来表示这个子查询。它是一个字典，包括了一个完整的查询语句中所具备的所有属性，可以是递归的。
- `many_values`: 在查询语句中，如果插入的是值，而不是查询结果，则用这个来表示它的值列表，因为在 `MySQL` 中可以同时插入多个值，则这里有可能是多个。
- `values`: 如果插入的是值，则它用来表示一行的插入内容，这是一个数组，每个元素是一个列的表达式，也是 `many_values` 数组的一个元素。而如果是一条更新语句，那么它表示的就是被更新的值表达式列表。
- `set_fields`: 用于表示更新语句的更新列的信息，这是一个数组，里面的每个元素对应被更新的一个列表表达式。

上面就是目前支持的语句中出现的标签说明，但是还有很大一部分是表达式的处理，在打印表达式时，每一个对象都有一个公共的 `Key`，名为 `type`，而针对不同的 `type`，其他的 `Key` 就不一定相同了，而具体的不同就不多叙述了，这里只给出支持哪些表达式，除 `type` 之外的其他信息，在使用过程中一试便知。

支持表达式类型

下面是目前支持的所有表达式的列表，仅列出了 `type` 的不同的值。

- `STRING_ITEM`: 字符串，有其他 `Key` 用来存储其具体值信息。
- `FIELD_ITEM`: 列信息，有其他 `Key` 用来存储具体对应的库、表及列名等。
- `FUNC_ITEM/COND_ITEM`: 逻辑运算信息，包括比较运算符、`AND`、`OR`、`ISNULL`、`ISNOTNULL`、`LIKE`、`BETWEEN`、`IN`、`NOT`、`NOW` 及其他自定义或内置函数等运算操作。
- `INT_ITEM`: 整数值表达式。
- `REAL_ITEM`: 符点数值表达式。
- `NULL_ITEM`: `NULL` 值表达式。
- `SUBSELECT_ITEM`: 子查询表达式。

- SUM_FUNC_ITEM: 集函数表达式。
- ROW_ITEM: 行表达式, 比如 `select * from t where (sno,name) = (select sno,name from t1)`, 这里 `where` 条件中等值表达式的左值就是这个表达式类型。
- DECIMAL_ITEM: DECIMAL 表达式类型。

以上就是目前所支持的表达式类型, 已经基本覆盖了所有常用的表达式。

最后要说明的是, 这里打印出来的信息, 已经不完全只是语法分析结束之后的信息了, 而是经过 Inception 加工过的。比如在查询语句中用到了子查询, 存在不同的上下文, 同时还使用了别名, 或者在使用列时没有指定其表名等这几种情况, Inception 都打印了每一列对应的库名表名, 这样打印出来的信息中, 就已经不存在没有定位 (找到其库名表名) 的列名了, 使用中更加友好准确。比如上面例子中就有这样的情况 (名为 `alisa_t` 的 `t` 表的别名)。也就是说, Inception 在内部已经将上下文的依赖分析完了, 有了自动化运维程序之后, 只管用就可以了。

53

Inception 设计

Inception 之源

Inception，来源于 MySQL，使用 Inception 的目的是要找到 MySQL 支持的 SQL 语句中，所有在 DBA 眼中，或者是在运维界公认的问题。MySQL 不能帮忙完成这个任务，所以只能是由 DBA 来自行完成了，但这是有门槛的，并不是很简单就可以完成。

MySQL 具有先天的优势，那就是它可以认识所有的 MySQL SQL 语句，所以还是可以想办法，让 MySQL 来帮我们一点忙。由于本人比较熟悉 MySQL 源代码，所以首先从改源代码的方面去考虑这个问题，因为采取这种方式，MySQL 已经完成了四方面的工作，如下。

- 服务协议，即 MySQL 协议，可以直接通过 MySQL 客户端及各种语言的 MySQL 接口来操作 Inception，这个工作量是非常大的，因此开发的工作量就大大减少了。
- 解析 SQL 语句：可以解析 SQL 语句的工具不少，有开源的、收费的，或自己实现等。但最熟悉 MySQL 的 SQL 语句，无疑还是 MySQL 本身，所以如果使用 MySQL 来改的话，那么首先这部分工作不需要做了，并且兼容性的问题也不需要考虑了，为开发工作解决了不少问题。
- 客户端连接监听：要想做一个服务程序，我们所熟知的基本的客户端-服务器模型的网络编程是需要的，无疑 MySQL 已经做得非常好了，因此我们不需要做任何事情就可以使用了。

- 结果集模型：Inception 最出彩的，让用户使用最方便的，无疑是它的结果集设计，而这一点也正是利用了 MySQL 内部结果集的设计方式。在 Inception 里面，只需要照猫画虎即可完成结果集的返回，非常方便。

从以上四点可以看到，Inception 所要用的的一些重要的模块实现，都在 MySQL 中已经实现得很好了，我们可以直接使用。所以在此基础上，只需要实现 Inception 的审核逻辑即可。从图 53.1 中可以看到 MySQL 与 Inception 实现的对比。



图 53.1

左边是 MySQL 的执行逻辑，客户端向服务器发送一个 SQL 语句的请求，MySQL 会分五个步骤对它进行处理，然后将结果返回给客户端。从上面的讲述来看，其中的语法分析、语义分析是 Inception 需要的部分，而其他部分都是不需要的，所以可以对 MySQL 进行大规模的裁剪，去掉不需要的部分。如果只是单纯的去掉，Inception 还不能满足我们的审核需求，所以需要再加上一些审核逻辑、执行逻辑等，就变成了图 53.1 中的右半部分，这样就把一件很大、很复杂的工作，变成了一个只需要实现自己业务逻辑的工作，极大地提高了效率，让 DBA 感受到了其强大的功能。

这样实现的 Inception，具有以下六方面的特点。

- 源自 MySQL：所有用到的东西，都是 MySQL 相关的，也都是我们所熟悉的东西，降低了使用门槛。

- 轻量级：相对于 MySQL，Inception 只剩下了分析审核相关的部分，而不需要在本地存储任何东西，变得非常轻量级，是一个无状态的软件，这进一步增强了 Inception 的友好性。
- 精准：这个特性是传统的审核方法无法比拟的。因为所有的分析结果，都来源于 MySQL 自身，那么针对 MySQL 的任何 SQL 语句来说，没有分析不到的表达式，也没有兼容性的问题，可以与 MySQL 完美对接，且可以通过内置的自定义规则，准确地找到不符合规则的问题。
- 支持接口丰富：因为直接使用了 MySQL 协议，包括 Python、PHP、C/C++ 等在内的很多语言，在 MySQL 方面都支持得很好，所以 Inception 也一样，可以像使用 MySQL 一样使用 Inception，接入门槛非常低。
- 结果集：通过结果集的方式，向应用程序展示非常丰富的内容，在使用 Inception 时，非常方便。另外，结构化的审核和执行结果，也让我们对错误或执行状态了如指掌。
- 上下文逻辑相关性：通过对线上表、库对象的缓存，Inception 做到了一个语句块内所有语句的上下文逻辑相关性，比如在一个语句块中，先创建一个库，马上使用，在这个库中创建一个表，然后再初始化这个表的数据等，诸如此类。此时这些库和表在线上还是不存在的，从上下文的角度来讲，在使用到这个对象时，前面已经创建过了，只是没有落地而已，但这对 Inception 审核来说，并不是问题，无疑极大简化了 DBA 及开发的使用难度，并且提高了效率。

Inception 执行流程

在设计好 Inception 的使用方式之后，所需要实现的就是审核、执行及备份的逻辑部分。现在已经知道，一个任务可以包含很多语句，这些语句通过 `inception_magic_start` 及 `inception_magic_commit` 两个语句包围起来，而中间部分，就是需要在线上真实执行的语句，它们之间可以有上下文逻辑关系，这都没问题。

至于在 Inception 拿到待审核语句之后，具体是如何操作的，先来看 Inception 的执行流程图，如图 53.2 所示。

可以看到，在 Inception 接到一个任务之后，主要分如下步骤来处理。

1. 初始化任务：因为 Inception 拿到的任务是一个很长的字符串，所以首先需要做的就是将这些语句按照规则分开。首先拿到的语句就是 `inception_magic_start`，这是 Inception 定义的合法的 SQL 语句，而这条语句是带有注释的，即数据库源信息部分，语句本身没有意义，但此时会处理很多事情，比如解析所有相关的选项，了解本次任务要做什么及在哪里执行、审核等，然后初始化本次任务处理的环境，比如获取相应的线上相关变量等，用来判断本次任务执行时的一些策略等。

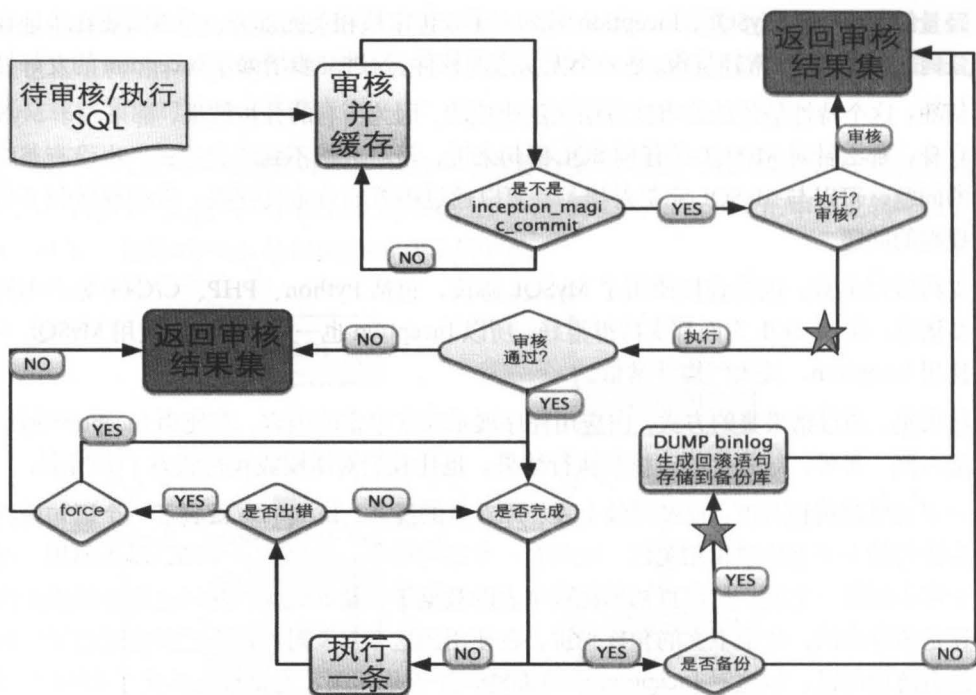


图 53.2

2. 审核: 在初始化环境之后, 就可以开始审核了。继续分析 SQL 语句, 通过语法分析, 将每一条语句分离出来, 每分离一条, 就先去审核, 找到所有相关的问题, 并且从线上将所有用到的库、表对象取出来, 用于辅助审核, 每审核完一条语句之后, 都将其缓存起来, 保存到 Inception 内部, 用来准备提交者需要的结果集。在审核到语句 `inception_magic_commit` 时, 说明本次任务已经审核完成。如果本次任务只是 `--enable-check`, 那么本次任务处理也就结束了, 将缓存好的结果集原样返回给提交者即可。而如果本次任务是 `--enable-execute`, 那么说明审核之后还要继续执行, 此时转到第 4 步。
3. 执行: 此时, Inception 会判断当前审核结果中, 是否发现了错误, 错误包括 2 (严重错误)、1 (警告)。如果是 2, 则直接将结果集返回给提交者, 里面包含了引起错误的描述; 而如果是 1, 则说明存在警告, 有些语句不符合规则, 此时就通过选项 `--enable-ignore-warnings` 来判断, 如果开启, 则说明要忽略这些警告, 开始执行。执行的过程, 就是将之前审核时缓存起来的语句列表, 按照顺序, 逐条地执行, 并且在执行每一条时, 都会将执行时对线上连接的 Binlog 模式设置为 ROW (为了生成回滚语句), 并且记录一下相关的 Binlog 位置、执行时间、thread id 等。如果执行过程中遇到了一些错误, 则此时 Inception 会判断有没有指定选项 `--enable-force`, 如果有则保存当前错误, 继续执行下一条, 如果没有, 就提交返回。

4. 备份：执行完成之后，就意味着本次任务已经完成，后面的工作，只是 Inception 的增值服务了，即我们所熟悉的备份并生成回滚语句的阶段。此时，Inception 会根据在执行时记录的 Binlog 位置，向线上数据库发送 Binlog Dump 请求。Inception 在拿到 Binlog 之后，通过多条件的过滤，如 Binlog 的大体位置、thread_id、事件类型等，找到属于 SQL 语句自己的 Binlog 之后，解析出来反拼为回滚语句，然后依次逐条地存储到备份库中，生成回滚语句的过程如图 53.3 所示。

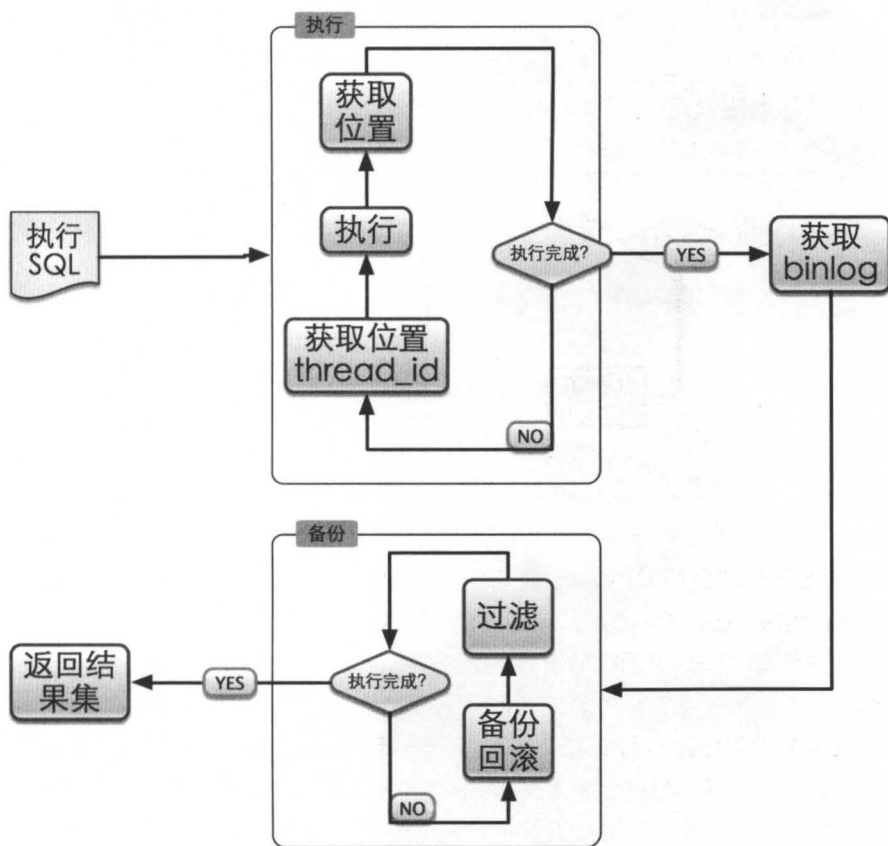


图 53.3

解析 Binlog 备份完成之后，此次任务的所有工作就都做完了，接下来的工作就是向提交者返回执行结果了。

5. 返回结果集：上面的过程，不管执行多长时间，提交者只能等待，因为 Inception 中间不会返回任何东西，这和 MySQL 执行一条语句是一样的。在所有操作都做完之后，或者执行出错了，才会将处理结果以结果集的形式发给提交者，提交者可以继续他的后续工作，而 Inception 此时的工作就是等待下一个任务。

/ 特邀撰稿 /

杜修文

现任Oracle公司MySQL技术顾问，负责大中华及亚洲地区MySQL的技术推广及项目的推动。同时也是台湾MySQL社区的发起人。

宋利兵

Oracle公司MySQL研发工程师，MySQL复制团队的成员，先后参加了MySQL 5.1以来的各个版本的开发工作。2016年参与了MySQL Group Replication的开发，期间研究了Group Replication的源代码，对Group Replication涉及的技术有透彻的理解。



本书交流平台

微信公众号: formysql

业内领袖寄语

在数据库世界里，MySQL是一个永恒的话题，不论是由于成本的原因，还是为了拥抱开源，或者能够更自由地做分布式扩展，很多互联网公司都把MySQL作为关系型数据库的首选。

去哪儿网在成立之初，也同样选择了MySQL。随着移动互联网和在线旅游行业的爆发，我们的访问量和数据量也越来越大，并且交易业务对数据完整性和一致性有了更加严苛的要求。在一次次遇到问题并且解决问题的过程中，我们的数据库工程师深入研究了MySQL的设计和源码，不断加深了对MySQL的理解，针对不同业务的需求场景选择了合适的数据库解决方案。

在开发服务业务的同时，我们的数据库工程师也不断地寻找提高工作效率的方法。当SQL审核成为越来越烦冗且无趣的工作时，我们自主研发了Inception。经过几个版本的迭代，Inception已经成为去哪儿网内部提升研发效率的利器之一，同时也规避了人工审核带来的主观性。现在我们已经将Inception部分开源，回馈到开源社区。

彦伟是一名优秀的数据库专家，ACMUG用户组创始人，在国内持续推广MySQL技术，并且获得Oracle ACED称号。在加入去哪儿网带领DBA团队后，解决了非常多的数据库难题，为业务的高速发展保驾护航，同时也培养出很多优秀的数据库工程师，目前他带领的DBA团队是去哪儿网最精干的团队之一。

甘泉 携程集团CTO，去哪儿网平台事业部CEO



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：徐津平
封面设计：侯士卿

上架建议：数据库

ISBN 978-7-121-31235-9



9 787121 312359 >

定价：119.00元